Adaptación de GDB para dar soporte a la arquitectura CoolFlux

---

Sistemas Informáticos
Facultad de Informática
Universidad Complutense de Madrid

DIRECTORES: LUIS PIÑUEL MORENO
JAVIER RESANO EZCARAY

AUTOR: CARLOS ZAFRA FDEZ. DE TEJADA

CURSO 2006-2007

Capítulo
1

Introducción

1.1 Resumen

El proyecto desarrollado ha consistido en portar el GNU Debugger para poderlo utilizar sobre una nueva arquitectura DSP (Procesador Digital de Señal). Concretamente, la arquitectura CoolFlux DSP que fue creada por Philips/NXP. Las principales características de esta arquitectura son su reducido consumo y su eficiencia en el procesado de señales de audio.

El GDB (GNU Debugger) es el depurador de uso más extendido y pertenece al proyecto GNU (GNU 's not Unix), que es de sobra conocido por sus múltiples herramientas de código abierto (por ejemplo, el intérprete de comandos Bash o el gestor de arranque Grub). Su éxito radica en su extensa funcionalidad y en el elevado número de sistemas que soporta. No obstante, dar soporte a un nuevo procesador supone un reto importante, especialmente si no se está familiarizado con su infraestratuctura y si se trabaja con un procesador en desarrollo, como es nuestro caso.

Como es bien sabido, un depurador es un software que permite al programador controlar la ejecución de un programa y examinar su estado. Gracias a éste puede descubrir fallos en tiempo de ejecución. En un sistema empotrado este proceso de depuración se divide habitualmente en dos partes. En el equipo de desarrollo (*host*) se ejecuta la parte de interfaz de usuario, que es la encargada de recibir los comandos de depuración, enviar las órdenes al equipo objetivo y visualizar la información que este último devuelve. En el equipo objetivo (*target*), se reciben estas órdenes, se controla la ejecución del programa y se suministra la información de estado solicitada. La comunicación entre ambos equipos se lleva cabo mediante uno o varios protocolos de comunicaciones.

Dar soporte a un nuevo sistema empotrado en GDB requiere por lo tanto: describir la arquitectura del sistema objetivo para poder comprender el estado en el que se encuentra; implementar los comandos de depuración mediante las órdenes básicas que el DSP es capaz de entender. Además, puesto que estas órdenes han de ser enviadas a través de uno o varios protocolos de comunicaciones, es necesario implementar los drivers que actuan de interfaz entre ellos.

Todos estos aspectos han sido tratados en este proyecto y el resultado es una versión completamente operativa del depurador GDB para CoolFlux, que permite incluso la depuración remota a través de Internet. Este entorno de depuración no sólo supera en funcionalidad y eficiencia al depurador oficial de la empresa Target, sino que además como efecto lateral de este proyecto se han corregido diversos errores de su compilador y se han descubierto comportamientos anómalos del propio procesador.

1.3 Summary

In this project I have ported the GNU Debugger to use it on DSP architecture. In fact, CoolFlux DSP architecture was created by Philips/NXP. CoolFlux DSP is an "Ultra Low Power DSP ", focus mainly on audio process.

GDB (GNU Debugger) is the debugger most known. It belongs to the GNU Project, which is well known for his multiple tools. For example, the command interpreter Bash or Grub, they belong to the GNU Project. The main particularity of this project is that it develops open source software.

A debugger is software that allows programmers to control the execution of a program. In fact, debugger helps programmers to discover failures and also it allows programmers try compiled code. GDB allows see what is happening 'inside' of another program, while this one is executed.

Programs can be debugged setting breakpoints, executing single step, observing the values of the variables, etc. The main characteristic is that, GDB can works with many different architectures.

Both main parts of this project, are the description implementation of CoolFlux architecture on GDB's files and the connection between the DSP and the computer.

This document, tries to explain on a detailed way, the necessary steps for the creation of a GDB version that allows remote debugging on any system with a CoolFlux core.

1.3 Objetivos

Actualmente existe un depurador oficial para CoolFlux realizado por la empresa Target, que también es la encargada de suministrar el compilador para esta arquitectura. Por lo tanto, a simple vista puede parecer absurdo realizar un nuevo depurador, sin embargo un análisis más detallado de las característcas y limitaciones del depurador oficial revela la necesidad una herramienta más potente, fiable y estandar. Los objetivos concretos que han ido guiando el desarrallo del proyecto han variado sustancialmente a mediada que este iba progresando, pero a grandes rasgos pueden resumirse del siguiente modo:

- Inicialmente, el objetivo final por parte de Philips/NXP era mucho menos ambicioso y consistía simplemente en crear una versión de GDBProxy. No obstante, una vez creada esta versión se comprobó que el rendimiento era insuficiente y que era necesario crear un nuevo depurador completo, aprovechando en la medida de lo posible el código e ideas del depurador oficial. Esto último no ha podido llevarse a cabo, ya que el software es propietario y la empresa Target se negó en todo momento a facilitar cualquier tipo de información sobre el mismo. La poca información que se ha obtenido proviene de los estudios de ingeniería inversa que se han realizado como parte de este proyecto.

- Una vez abandonada la idea del GDBProxy, el objetivo perseguido fue desarrollar una versión con funcionalidad mínima del GDB, implementando los drivers necesarios para la comunicación mediante TCP/IP.

- Una vez logrado el objetivo anterior, se trató de ampliar la funcionalidad portando elementos como el backtrace, análisis de prólogo, frames, etc.

- En paralelo con lel objetivo anterior, se ha tratado de mejorar la eficiencia de los drivers e insertar nuevos protocolos de comunicación (USB, puerto paralelo, etc). Asimismo, se ha trado de integrar el depurador con entornos de desarrollo usuales como DDD o Eclipse.

Como se podrá comprobarse más adelante estos objetivos han sido cubierto prácticamente en su totalidad.

1.4 Estructura

1.4.1 Esquema

A continuación se muestra un esquema aproximado que recoge como se conectan cada una de las partes del proyecto.

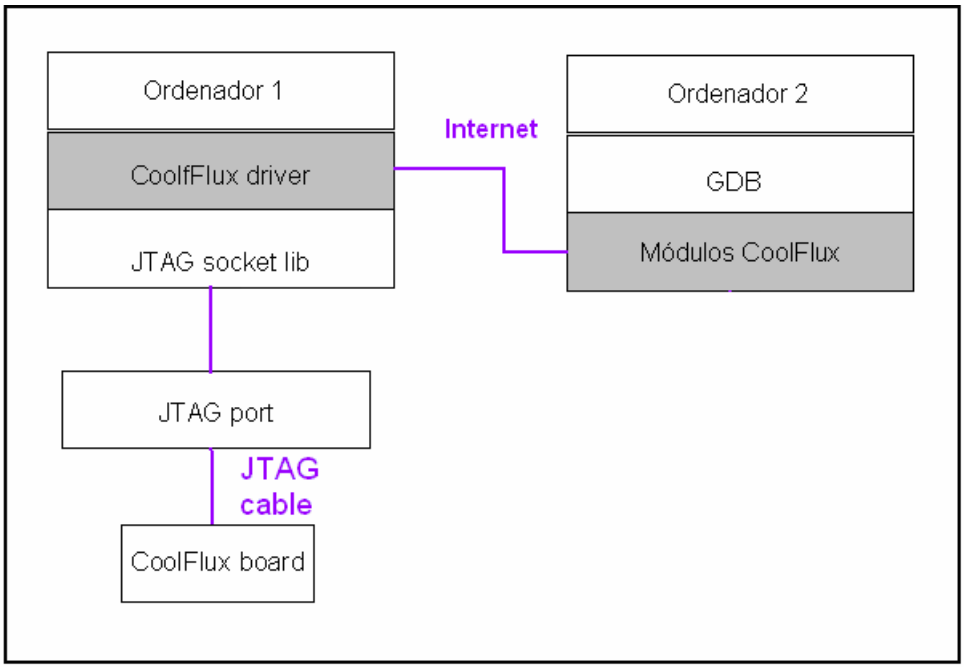


Figura 1: Esquema general del proyecto

Las partes en gris, son las partes que constiuyen el grueso del proyecto, y a continuación las describimos someramente.

CoolFlux Driver

Este módulo es el encargado de la comunicación con el sistema objetivo y normalmente consta de 2 partes. La primera (driver TCP/IP) se encarga de la comuncicación mediante sockets con el equipo remoto en el que se ejecuta el GDB y actúa de interfaz con la otra parte. La segunda parte (driver paralelo/JTAG) realiza la comunicación con el sistema empotrado a través del puerto paralelo y JTAG.

Módulos de CoolFlux

Aquí es donde se concentraron la mayor parte de nuestros esfuerzos y donde se ha invertido la mayor parte del tiempo. A grandes rasgos estos módulos permiten describir la arquitectura en terminos comprensibles por GDB y definir los comandos de depuración en función de las órdenes que módulo interno de depuración del CoolFlux es capaz de entender. Dichos módulos se encuentran recogidos principalmente en los dos siguientes ficheros:

- El archivo *target-CoolFlux*, que se encarga de gestionar la comunicación entre GDB y CoolFlux. Este archivo, convierte los comandos de GDB en comandos JTAG, y según el medio que se ha elegido los envía al CoolFlux.

- El archivo *CoolFlux-tdep*, en el que se definen todas las funciones y algoritmos, que son necesarios para dar soporte a todas las funcionalidades de GDB. Por poner un ejemplo, dentro de este archivo se implementan los algoritmos de análisis de marco de pila, análisis del prólogo, call functions, etc… Por supuesto algunas de estas funcionalidades no es obligatorio implementarlas, sin embargo, suelen ser bastante útiles.

Organización de la memoria

Esta memoria se ha dividido 5 capítulos. El presente capítulo introduce los objetivos y describemente la estructura del proyecto. El capítulo 2 describe la arquitectura CoolFlux con cierto detalle e introduce algunas nociones sobre el protocolo de depuración JTAG. El capítulo 3 describe la estructura del depurado GDB incidiendo en aquellos aspectos que tienen que ver con el soporte de nuevas plataformas. El capítulo 4 detalla la implementación llevada a cabo y finalmalmente, el capítulo 5 recoge las conclusiones y el trabajo futuro.

Se han añadido 3 apéndices que recogen cierta información que puede resultar de interés aunque no forman parte esencial del proyecto. Al apéndice A, describe las MACROS del Target-Side de GDB. En el apéndice B se incluye una plantilla del fichero “remote-CoolFlux.c” que sirve de base para portar nuevas arquitecturas al GDB. El uso de esta plantilla, generada como efecto lateral del proyecto, reduce significativamente el tiempo de desarrollo. Finalmente, el apéndice C, recoge parte del proyecto de otro estudiante que ha sido empleado para dotar al nuevo depurador de capacidad de desensamblado para esta arquitectura. En este apéndice se describe el uso de una gramática CGEN (Cpu tools GENerator) para generar los opcodes de GDB.

Capítulo 2

Arquitectura CoolFlux DSP

2.1 Visión General

CoolFlux DSP es una arquitectura de doble MAC, y doble Harvard capaz de realizar dos MACs, dos operaciones de memoria y dos actualizaciones de punteros por ciclo, haciendo esta arquitectura altamente eficiente para aplicaciones de cálculo intensivo.

La arquitectura CoolFlux DSP es una arquitectura load/store:

- Todos los operandos en memoria tienen que ser movidos a los registros
- Todas las unidades de ejecución toman sus valores de entrada de los registros y escriben sus resultados también en los registros
- Todos los resultados producidos tienen que ser movidos a la memoria

Las figuras de las siguientes páginas (Figura 2 y 3) dan un esbozo de la arquitectura CoolFlux DSP. En ellas se muestra:

- La unidad de control con los registros pc, sr, isr, lr y ilr y 64 Kwords de 32 bit de memoria de programa
- Las unidades de direccionamiento X y Y, contienen:
  - La XAGU y la YAGU, la cuales se encargan de la generación de direcciones
  - Los registros xptr, xstep, xmod y sp
  - Los registros yptr, ystep, ymod
  - 64 Kword de 24-bit Memoria-X, Memoria-Y y Memoria-I/O
- El bloque DMA
- Los 24-bit Xbus y Ybus
- Los registros X y Y de 24-bit(x0, x1, y0, y1)
- El acumulador A y B de 56-bit (a0, a1, b0, b1):
  - Los multiplicadores de 24x24 bit XMUL y YMUL
  - La unidad pre-suma/resta del multiplicador XMUL
  - La XALU y la YALU
- Las dos unidades RSS (Redondeo, saturación y selección)

- Los bloques de conversión de los distintos anchos de palabra

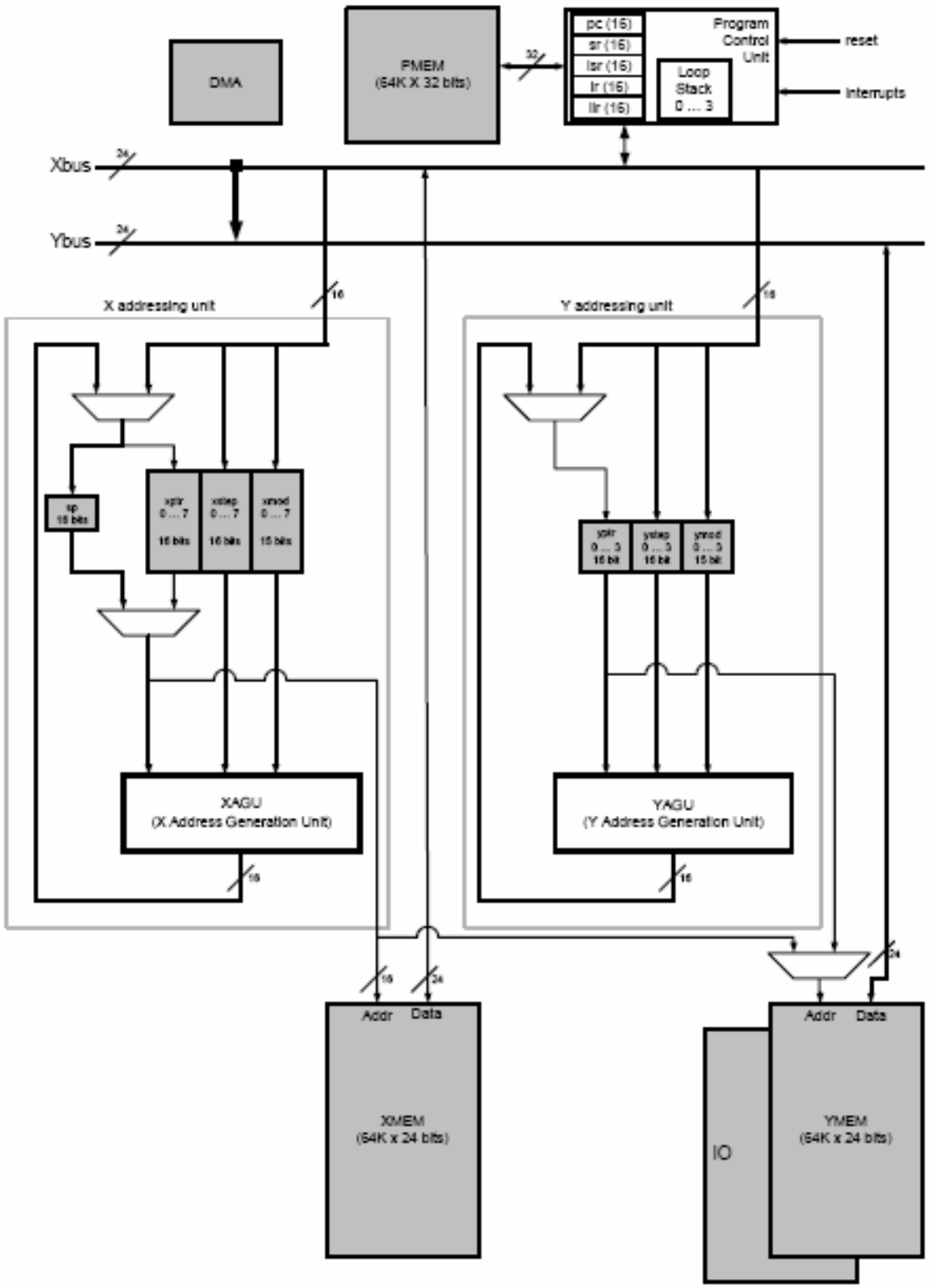


Figura 2: Unidades de direccionamiento, buses, memorias, DMA y unidad de control

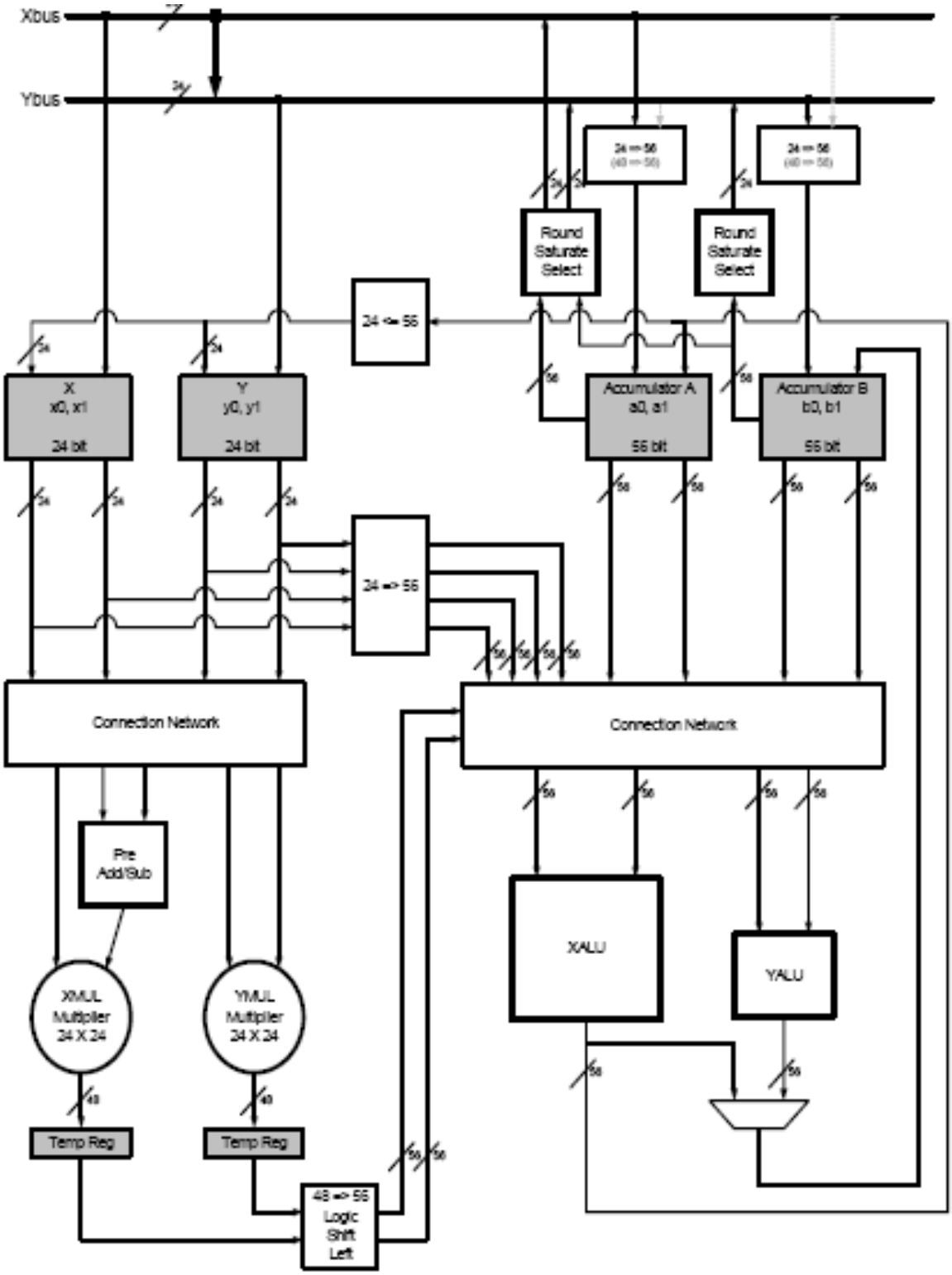


Figura 3: Ruta de datos y unidades de redondeo, saturación y desplazamiento

2.2 Registros

Todas las unidades de ejecución toman sus valores de entrada de los registros y escriben sus resultados también en los registros. Los accesos a memoria son considerados como operaciones. Una operación de store toma los datos de los registros y los escribe en memoria, mientras que una operación de load lee los valores desde la memoria y los guarda en los registros.

Los registros del CoolFlux DSP están organizados en "register files" o ficheros de registros. Todos los registros en un "register file" tienen el mismo nombre pero distinto índice. Las operaciones trabajan sobre registros individuales. La razón para esto, es que todos los registros en un "register file" pueden ser usados a la vez por la misma operación. Los registros individuales deben ser usados como fuente y destino en operaciones reales.

2.3 La ruta de datos

La ruta de datos consiste en:

- 4 "register files" A, B, X e Y conectados a las unidades aritméticas
- Dos multiplicadores XMUL y YMUL (24x24 bits)
  - Con una unidad de suma/resta antes del multiplicador de XMUL
- 2 ALU, XALU y YALU:
  - La XALU ejecuta operaciones aritméticas simples y operaciones XMAC
  - La YALU ejecuta operaciones aritméticas en paralelo y operaciones YMAC

La arquitectura CoolFlux DSP claramente muestra simetría. Sin embargo, las llamadas operaciones-X (usando la XALU y el XMUL) son mas complejas.

2.3.1 Multiplicadores

El CoolFlux DSP contiene 2 unidades de multiplicación, XMUL y YMUL, ambas de 24x24 bits. Ambas toman sus operandos de los "register files" X e Y. La unidad XMUL puede ser usada para multiplicar signed/signed, unsigned/signed y unsigned/unsigned. La unidad YMUL soporta multiplicaciones signed/signed.

La unidad XMUL es precedida por un pre-sumador/restador. Esta unidad puede ser usada en 3 modos: *pass*, *add* y *subtract*. Para multiplicaciones normales se suele usar el modo pass, en el cual solo se transfiere uno de sus operandos desde la entrada hasta la salida, sin hacer ninguna operación. Esta salida es entonces multiplicada por la unidad XMUL con otro registro XY. En el caso de que se use el modo add/subtract, se sumaran o se restaran dos registros XY y la salida será multiplicada en XMUL por un tercer registro XY.

En todos los casos el resultado de la operación será de 48-bit. El resultado es guardado en un registro temporal y en el próximo ciclo se usara como operando para la acumulación en una de las ALU (XALU para resultados de XMUL, YALU para

resultados de YMUL). Antes de ser usado en las ALUs es necesario convertir el número a 56-bit.

Los multiplicadores son siempre accedidos vía operaciones MAC, las cuales necesitan 2 ciclos para ser ejecutadas. En el primer ciclo la multiplicación es ejecuta en XMUL o YMUL. En el segundo ciclo el resultado es acumulado a través de XALU o YALU. En cada nuevo ciclo, una nueva operación MAC puede ser ejecutada, por lo tanto la multiplicación corre en paralelo con la acumulación de la anterior operación. Esto significa que las n operaciones MAC consecutivas solo tardan n+1 ciclos.

2.3.2 Unidades Aritmético-Lógicas (XALU y YALU)

La XALU y la YALU siempre operan con operados de 56-bit, y generan resultados de 56-bit. Las unidades toman sus operandos desde los registros A, B, X, Y o desde los registros temporales a la salida de los multiplicadores. Todos los operandos son lógicamente convertidos a 56 bits. Los resultados de la XALU pueden ser almacenados en los registros A, B, X e Y; los resultados de la YALU pueden ser almacenados solo en el registro B. El resultado es convertido a 24 bits cuando es escrito en un registro X o Y.

Las posibles operaciones para la XALU son:

- Sumar, restar, sumar y dividir por 2, restar y dividir por 2, suma reducida, resta reducida
- And, Or, Xor
- Divide step
- Min, max
- Negación, exponenciación, valor absoluto, extensión de signo
- Desplazamientos, desplazamientos reducidos
- Operaciones de comparación
- Cargar un valor inmediato

Adicionalmente, algunas de estas operaciones pueden ser ejecutadas de manera condicional.

La YALU soporta: suma, resta, sumar y dividir por 2, restar y dividir por 2.

2.4 Buses y unidades RSS

2.4.1 XBus e YBus

Los movimientos entre los registros, y también entre la memoria y los registros son esenciales porque la arquitectura del CoolFlux DSP es load/store.

Los dos buses centrales son los encargados de mover los datos, en paralelo con las operaciones en la ruta de datos. Estos buses proporcionan una conexión entre todos los registros, y entre los registros y las memorias. Los tipos de movimientos permitidos son:

- Desde un registro a otro registro
- Desde un registro a memoria (store)
- Desde memoria a un registro (load)

Los buses soportan movimientos en modo individual y paralelo. En un movimiento individual, un dato de 24-bit es movido en un solo ciclo de procesador. Con un movimiento en modo paralelo, dos datos de 24-bit son movidos en un solo ciclo de procesador usando los dos buses al mismo tiempo.

En cada movimiento el valor fuente es convertido a 24-bit. Estos 24-bit son transferidos al registro destino, donde serán convertidos al tamaño del registro destino.

Los buses de 24-bit son suficientes para hacer todos los movimientos de 8-bit, 15-bit, 16-bit and 24-bit sin ninguna complicación, sin embargo transferir desde y a un acumulador de 56-bit requiere un tratamiento especial que se explica a continuación.

2.4.2 Mover datos a un acumulador

El acumulador puede ser el destino de dos maneras distintas:

- El acumulador completo (a0, a1, b0, b1)
- Un sub-registro del acumulador (ao0, ao1, ah0, ah1, al0, al1, bo0, bo1, bh0, bh1, bl0, bl1)

En el caso de que un sub-registro sea el objetivo, el tamaño del dato deberá ser de 8 o 24 bits.

El acumulador completo acepta operandos fuentes producidos por:

- En formato de 24 bit
- En formato de 48 bit

Al mover 24-bit (‘=’ asignación), los 24 bits son convertidos a 56 bits extendiendo el signo a los 8 bits más significativos y rellenando con 24 ‘0’s el LSB.

Al mover 48-bit (‘=.l’ asignación), el Xbus y el Ybus son combinados para mover un dato de 48-bit. Estos 48-bit son convertidos a 56 bits extendiendo el signo con 8 bits. Un movimiento de 48-bit es solo posible entre dos acumuladores, o desde la memoria XY a uno de los acumuladores.

2.4.3 Mover datos desde un acumulador

El acumulador puede generar operandos Fuentes, de dos maneras:

- Como un acumulador completo (a0, a1, b0, b1)

- A través de uno de sus sub-registros (ao0, ao1, ah0, ah1, al0, al1, bo0, bo1, bh0, bh1, bl0, bl1)

En el caso de que un sub-register sea la fuente de una operación de movimiento, el tamaño del dato deberá ser de 8 o 24 bits.

El acumulador completo genera operandos fuentes, de dos maneras:

- En formato 24-bit
- En formato 48-bit

Al mover 24-bit ('=' asignación), el valor de 56-bit es convertido a 24 bits por medio del redondeo y la saturación de la unidad RSS y es transferido registro destino, por medio de uno de los dos buses.

Al mover 48-bit ('=.l' asignación), el Xbus y el Ybus son combinados para mover un dato de 48-bit, en un solo ciclo de procesador. Las 56-bit fuentes son convertidos a 48 bits a través de la lógica de saturación de la unidad RSS. Un movimiento de 48-bit solo es posible entre dos acumuladores, o desde uno de los acumuladores a la memoria XY.

2.4.4 Redondeo, Saturación y Selección (RSS Unit)

El CoolFlux DSP tiene dos unidades RSS, una para los acumuladores A y otra para los acumuladores B. Estas unidades convierten los valores mantenidos en 56-bit en los acumuladores a 24-bit o 48-bit (depende del tipo de movimiento) y son usados cuando los 56-bit de un acumulador son un operando fuente de una operación de movimiento. En el caso de que un sub-registro sea usado como operando fuente, la unidad RSS seleccionará la parte correcta (overflow, high, low) del acumulador.

En el caso de que el acumulador complete sea el operando fuente, el valor del mismo necesita ser convertido de 56 bits a 24 o 48 bits lo cual requiere saturación y redondeo. Ambas son controladas por las unidades RSS, las cuales pueden operar en diferentes modos.

Los modos para el redondeo son:

- Truncar, sesgo simple
- Truncar, magnitud y signo
- Redondeo, sesgo simple
- Redondeo sesgo-balanceado

Los modos para la saturación son:

- Saturación
- Wrapping

Los modos de operación son iguales para ambas unidades RSS. Además están controladas por los bits *sr.b, sr.r y sr.s* en el *sr* status register.

2.5 Memorias E/S

El CoolFlux DSP es una arquitectura Harvard dual y tiene separadas las memoria X e Y. Cada memoria tiene 64 Kword de 24 bits. En un solo ciclo de procesador cada memoria puede ser accedida separadamente, por lo que 2 datos pueden ser leídos o escritos en un solo ciclo. Hay un modo especial en el cual la memoria X e Y son combinadas lógicamente en una sola memoria XY que almacena y carga datos de 48-bit.

2.5.1 Memoria de datos (X,Y)

Normalmente los accesos a ambas memorias son en paralelo para DSP conseguir suficiente ancho de banda para aplicaciones de cómputo intensivo.

2.5.2 Modelo de doble precisión

Hay instrucciones de doble precisión (load/store), que pueden mover un dato de 48-bit (long) a/desde el sub-sistema de memoria. Para estas instrucciones las memorias X e Y, y ambos buses Xbus e Ybus son combinados (usados en paralelo). Las direcciones de memoria correspondientes necesitan estar reservadas puesto que, en este modo, ambas memorias cogen los datos de la misma dirección. Esta memoria se llama memoria XY, usa la unidad X de generación de direcciones y tiene las mismas posibilidades que la memoria X.

2.5.3 Espacio de memoria I/O

El I/O para el CoolFlux DSP esta mapeado en memoria. El I/O-space tiene 64 Kwords, 24-bit. La memoria I/O puede ser direccionada como la memoria Y, usando la unidad de generación de direcciones Y para generar dichas direcciones.

2.6 Unidades de generación de direcciones

Los siguientes modos de direccionamiento están permitidos:

- Directo
- Indirecto con post-modificación
- Indexado con puntero de pila

El direccionamiento indirecto esta soportado gracias a 2 “register files” de punteros: XPTR, YPTR. Estos se encuentran dentro de las unidades de generación de direcciones. El “register file” XPTR contiene 8 punteros (xptr0 … xptr7); El “register file” YPTR contiene 4 punteros (yptr0 … yptr3).Cuando las memorias son accedidas indirectamente por uno de estos punteros, el valor de estos punteros puede ser actualizado (post-modificado) en paralelo con el acceso. Las operaciones para la actualización se ejecutan en la XAGU e YAGU. La operación de post-modificación es especificada como parte del acceso a memoria. Las posibles operaciones de post-modificación son:

- nop
- incremento

- decremento
- incremento en 2
- decremento en 2
- suma y step registro
- resta y step registro
- y el modo especial que se usa para:
  - Buffer cíclico
  - Bit-reverse

Además de que las actualizaciones son parte de la sintaxis del direccionamiento indirecto, hay un número de operaciones específicas que son ejecutadas en XAGU e YAGU.

Las operaciones que se ejecutan en XAGU e YAGU son ejecutadas en paralelo con las operaciones de movimiento de los buses Xbus e Ybus, los accesos a memoria, y las operaciones en la ruta de datos (XMUL, YMUL, XALU e YALU).

2.6.1 Direccionamiento Indexado (Puntero de Pila)

La unidad de direccionamiento X contiene un puntero de pila sp. Este puntero de pila da un pleno control de la pila a la memoria X. El Direccionamiento indexado usa el puntero de pila como puntero base, más un offset respecto del puntero de pila. En este caso, el puntero de pila no se actualiza automáticamente, sin embargo, hay un gran número de operaciones para la actualización del puntero de pila.

2.7 Unidad de control

Esta unidad controla el contador de programa, el status register y los registros usados para interrupciones y subrutinas. Controla el flujo del programa e implementa los saltos, las llamadas a subrutinas, *hardware loops* o bucles hardware, y las interrupciones. También guarda el estado del status register

Implementa 4 registros que son directamente accesibles por el código:

Nombre	**Nombre Real**	**Tamaño**	**Descripción**
sr	Status Register	16 bits	Contiene el control de la máquina y el status word
lr	Link Register	16 bits	Contiene la dirección de retorno después de una instrucción call
ilr	Interrupt Link Register	16 bits	Contiene la dirección de retorno después de una

			interrupción
isr	Interrupt Status Register	16 bits	Es usado para salvar el valor del status register sr durante una interrupción

Tabla 1: Registros de la unidad de control

El status register contiene tres flags (zero, negative y overflow) y un número de bits de control, los cuales controlan las operaciones del CoolFlux DSP. Tres bits controlan las acciones de ambas unidades RSS (sr.b, sr.r, sr.s). El bit sr.ie se encarga de activar o desactivar las interrupciones.

2.7.1 Hardware Loop Stack

Hardware loops están implementados a través de una pila de hardware loop. La pila tiene una profundidad de 4, lo cual quiere decir que solo pueden ejecutarse 4 bucles mediante hardware. Cuando el bucle es iniciado, la dirección de inicio y de finalización del código, y el número de veces que el bucle tiene que se ejecutado es almacenado en la pila. Cuando un bucle termina, la información del bucle es extraída de la pila.

La pila esta compuesta por:

- Cuatro direcciones de comienzo de 16-bit
- Cuatro direcciones de finalización de 16-bit
- Cuatro contadores de bucle de 12-bit (1 … 4096)

2.8 Interfaz Hardware

El CoolFlux DSP tiene una gran cantidad de facilidades para la I/O. Entre ellas se incluye mapeo de periféricos en memoria, tres interrupciones hardware, un controlador DMA para acceder a las memorias de datos y otro controlador para acceder a la memoria de programa (P).

2.8.1 DMA

El CoolFlux DSP tiene un esquema de acceso a memoria arbitrado para todas las memorias que soporta “hand shaken” DMA. El controlador DMA tiene un espacio de direcciones único que puede emplearse en la memoria X o en la Y. El puerto de programa del DMA tiene acceso a la memoria P. Acceder a la memoria X, Y y P es posible cuando el CoolFlux DSP esta ejecutando un programa. El CoolFlux DSP puede soportar caches de datos e instrucciones a través del uso de “core hold signals”.

2.9 Instrucciones y operaciones

Hay una clara distinción entre una instrucción y una operación del CoolFlux DSP. Una instrucción es de 32-bit. Las instrucciones se leen desde la memoria de programa durante la etapa fetch del procesador. Cada instrucción esta compuesta de 1,2,3 o 4 operaciones. Cada operación define una acción para cada parte del CoolFlux DSP. Las operaciones de una única instrucción se lazan en paralelo, durante el mismo ciclo de procesador.

2.9.1 CoolFlux DSP Operaciones

Operaciones simples

Existen tres operaciones básicas:

- Single arithmetic operation (SA)
  - Ejecutadas en la ruta de datos (YMUL, XMUL, XALU, YALU)
- Single move operation (SM)
  - Transferencias de datos, por medio del Xbus y del Ybus, desde un registro hasta otro registro, desde la memoria a los registros (load) o desde los registros a la memoria (store)
  - Incluye operaciones de actualización de los punteros, las cuales son ejecutadas en la XAGU y la YAGU
- Long operation (LO):
  - Ejecuta una sola operación la cual afecta a todo el procesador (instrucciones de control de flujo) o las cuales requieren un inmediato de tipo long (load/store)

NOTACION:

SA // single arithmetic operation

SM // single move operation

LO // long operation

Operaciones en paralelo

Las operaciones en paralelo se definen de dos maneras distintas:

- Parallel arithmetic operation (PA)
  - Es la combinación de operaciones 2 single arithmetic (SA) las cuales son ejecutadas en paralelo.
  - Existen reglas y limitaciones, por la cuales algunas operaciones SA pueden ser combinadas para formar una operación PA, mientras que en otras ocasiones no es posible (no hay unidades aritméticas disponibles, etc.)
  - La primera operación SA es ejecutada en XMUL y XALU, la segunda operación SA es ejecutada en YMUL y YALU
- Parallel move operations (PM)
  - Es la combinación de 2 operaciones single move (SM) que son ejecutadas en paralelo

- Hay reglas y limitaciones por las cuales, las operaciones SM no pueden ser combinadas (No todos los registros pueden ser usados como destino)
- La primera operación usa el Xbus, mientras que la segunda usa el Ybus

NOTACIÓN

SA1, SA2 // parallel arithmetic operation

SM1, SM2 // parallel move operation

2.9.2 CoolFlux DSP Instrucciones

Instrucciones Aritmethic-Move (AM)

Esta instrucción combina una operación aritmética simple o paralela con una operación de movimiento simple o paralela en una instrucción. Cuando una de las 2 partes (aritmética o movimiento) no pueda ser rellenada, una operación nop debe ser usada explícitamente.

NOTACIÓN

SA, SM ; // single arithmetic – single move

SA, nop ; // single arithmetic only

nop, SM ; // single move only

SA1, SA2, SM ; // parallel arithmetic – single move

SA1, SA2, nop ; // parallel arithmetic only

SA, SM1, SM2 ; // single arithmetic – parallel move

nop, SM1, SM2 ; // parallel move only

SA1, SA2, SM1, SM2 ; // parallel arithmetic – parallel move

Instrucciones Move-Move

Esta es una optimización especial. Esta instrucción combina 2 intrucciones AM consecutivamente en una sola instrucción move-move. Esto es posible cuando la parte aritmética es una operación nop. Esta instrucción necesita de 2 ciclos para ser ejecutada. El procesador traduce esta instrucción a su original compuesta por 2 instrucciones AM, poniendo una instrucción nop en la parte aritmética. Las operaciones de movimiento pueden ser simples o paralelas. Gracias a este mecanismo se ahorra una fase de fetch.

NOTACIÓN

mm(Move1 | Move2) ;

Esta instrucción se transforma en las dos siguientes instrucciones AM que serán ejecutas en los dos siguientes ciclos de reloj:

nop, Move1 ;

nop, Move2 ;

Las operaciones Move1 y Move2 pueden ser simples o paralelas, independientemente.

Move1 = SM or SM1,SM2

Move2 = SM or SM1,SM2

2.10 Instrucciones Multi-Ciclo y Delay-Slots

El CoolFlux DSP tiene un pipeline de 3 etapas: fetch, decode and execute. Como consecuencia, todas las instrucciones de control de flujo, (instrucciones que cambian el pc "program counter"), necesitan ejecutarse en múltiples ciclos, porque el pipeline necesita ser vaciado y rellenado en cada cambio del program counter pc.

2.10.1 Instrucciones Multi-Ciclo

La siguiente tabla muestra como funcionan las etapas del pipeline para instrucciones que no son de control de flujo: La dirección de la próxima instrucción es la dirección de la dirección actual + 1, y el pipeline funciona como se muestra a continuación.

Cycle n	Cycle n+1	Cycle n+2	Cycle n+3	Cycle n+4
fetch instruction i+1	fetch instruction i+2	fetch instruction i+3	fetch instruction i+4	fetch instruction i+5
decode instruction i	decode instruction i+1	decode instruction i+2	decode instruction i+3	decode instruction i+4
execute instruction i-1	**execute instruction i**	execute instruction i+1	execute instruction i+2	execute instruction i+3

Tabla 2: Ejemplo de ejecución

Cuando la instrucción i es una instrucción de control de flujo (un salto, una llamada, un return desde una subrutina, un salto condicional ...), la siguiente instrucción que será ejecutada después de esta instrucción no tiene porque ser la instrucción i+1, por lo que el pre-fetch falla. Otra instrucción (en la dirección k, el objetivo del salto) ha tenido que ser decodificada, lo cual lleva algunos ciclos extra:

Cycle n	Cycle n+1	Cycle n+2	Cycle n+3	Cycle n+4
fetch instruction i+1	fetch instruction i+2	fetch instruction k	fetch instruction k+1	fetch instruction k+2
decode instruction i	decode instruction i+1		decode instruction k	decode instruction k+1
execute instruction i-1	**execute instruction i**			execute instruction k

Tabla 3: Ejemplo de ejecución

El procesador tiene que esperar hasta que la instrucción sea buscada y decodificada, esto lleva algunos ciclos. En la tabla anterior, vemos que tarda exactamente tres ciclos en completar la operación.

2.10.2 Delay Slots

Hay una alternativa distinta a parar el pipeline en el caso de las instrucciones de control. Esta solución consiste en usar los delay slots. Existen multitud de casos en los que se pueden usar los delay slots para ejecutar otra instrucción y así incrementar el rendimiento. A continuación se dan los datos de las diferentes instrucciones multiciclo, indicando el número total de ciclos por instrucción y el número de delay slots.

Subroutine call / return		
Operation	# Cycles	# Delay slots
dcall *addr*	2	1
dcall *XPTR*	3	2
dreturn	3	2

BRANCHING		
Operation	# Cycles	# Delay slots
goto *addr*	2	None
dgoto *addr*	2	1
dgoto *XPTR*	3	2

Conditional BRANCHING			
Operation	Condition-code	# Cycles	# Delay slots
if (CC) goto *addr*	CC = false	1	None
	CC = true	3	None
if (CC) dgoto *addr*	CC = false	1	None
	CC = true	3	2
if (CC) dgoto *XPTR*	CC = false	1	None
	CC = true	3	2

Interrupt handling		
Operation	# Cycles	# Delay slots
swi imm3u	2	none
ireturn	3	none
idreturn	3	2
sr.ie = '1'	3	none

Interrupt handling		
Operation	# Cycles	# Delay slots
swi imm3u	2	none
ireturn	3	none
idreturn	3	2
sr.ie = '1	3	none

Tabla 4: Delay Slots y número de ciclos de cada instrucción

2.11 Jtag

JTAG, un acrónimo para Joint Test Action Group, es el nombre común utilizado para la norma IEEE 1149.1 titulada "Standard Test Access Port and Boundary-Scan Architecture" utilizada para testear PCBs utilizandoel método boundary scan.

JTAG se estandarizó en 1990 como la norma IEEE 1149.1-1990. En 1994 se agregó un suplemento que contiene una descripción del *boundary scan description language*

(BSDL). Desde entonces, esta norma fue adoptada por las compañías electrónicas de todo el mundo. Actualmente, Boundary-scan y JTAG son sinónimos.

Diseñado originalmente para circuitos impresos, actualmente es utilizado para la prueba de sub-módulos de circuitos integrados, y es muy útil también como mecanismo para depuración de aplicaciones empotradas, puesto que provee una puerta trasera hacia dentro del sistema. Cuando se utiliza como herramienta de depuración, un emulador en circuito que usa JTAG como mecanismo de transporte permite al programador acceder al módulo de depuración que se encuentra integrado dentro de la CPU. El módulo de depuración permite al programador corregir y controlar la ejecución del programa y consultar el estado interno de la CPU.

2.11.1 Esquema General

Una interfaz JTAG es una interfaz especial de cuatro o cinco pines agregadas a un chip, diseñada de tal manera que varios chips en una tarjeta puedan tener sus líneas JTAG conectadas en daisy chain, y una sonda de testeo necesite conectarse solamente a un solo "puerto JTAG" para acceder a todos los chips en un circuito impreso. Los pines del conector son:

- TDI (Entrada de Datos de Testeo)
- TDO (Salida de Datos de Testeo)
- TCK (Reloj de Testeo)
- TMS (Selector de Modo de Testeo)
- TRST (Reset de Testeo) es opcional

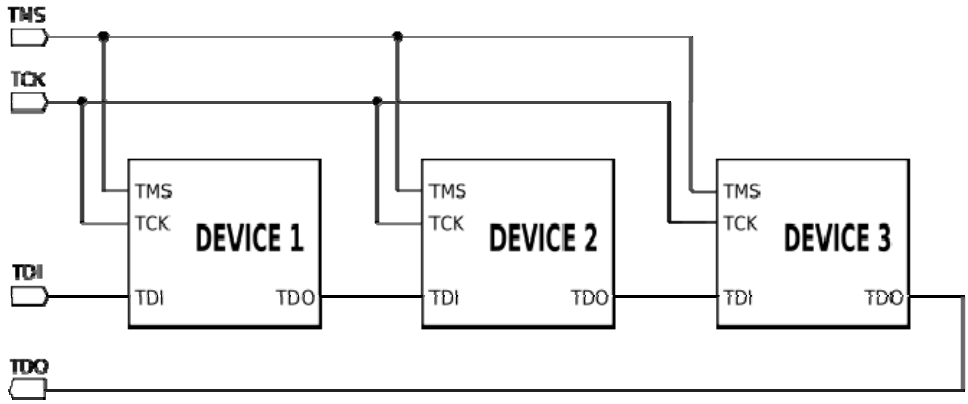


Figura 4: JTAG Chain

Ya que posee una sola línea de datos, el protocolo es obviamente en serie. La señal de reloj está en el pin TCK. La configuración del dispositivo se realiza manipulando un bit de estado y al mismo tiempo por el pin TMS. Cada bit de datos entrante/saliente se transfiere en cada pulso de reloj por la línea con pines TDI/TDO, respectivamente. Se pueden cargar diferentes modo de instrucción para leer el chip ID, pines de entrada/salida, manipular funciones del chip, funciones de bypass y de boundary scan.

El pin TRST es una señal opcional activa a baja para reseteo o reinicio de la prueba lógica (por lo general asíncrona, pero que a veces sincronizada con el reloj, dependiendo del chip). Si no se dispone de dicho pin, la prueba lógica puede reiniciarse mediante una instrucción reset.

Existen productos de consumo que tienen un puerto JTAG integrado, por lo que las conexiones están a menudo disponibles en la PCB como parte de la fase de prototipado del producto.

Como se ha mencionado anteriormente JTAG, permite una comunicación serie con los elementos internos del PCB/Chip empleados para el testeo y/o depuración. La interpretación de los datos que se transmiten dependen de dichos elementos. Para más información acerca de los comandos se puede consultar información más detallada en los manuales de CoolFlux, referencias 3 y 4.

Capítulo 3

GDB

3.1 Estructura General

GDB se compone de tres subsistemas: La Interfaz del Usuario, el Manipulador de Símbolos (Symbol Side) y el Manipulador del Sistema Objetivo (Target Side).

La división target side/ symbol side no es formal, hay muchas excepciones. Por ejemplo, el núcleo del GDB se compone de elementos simbólicos y elementos del sistema objetivo.

3.1.1 Symbol Side

El symbol side se compone de objetos lectores de archivos o ficheros, interpretes de información de depuración, el manejo de la tabla de símbolos, el análisis del código fuente e impresión de tipos y valores.

La parte simbólica de GDB puede entenderse como cada cosa que puedes hacer en GDB sin tener un programa en ejecución. Por ejemplo, puedes mirar los tipos de variables y evaluar muchas clases de expresiones.

3.1.2 Target Side

El target side se compone del control de ejecución, el análisis del marco de la pila o "stack frame", y la manipulación física del sistema objetivo o target.

El sistema objetivo del GDB es el responsable de la manipulación de los datos (bits/bytes). Aunque es conveniente disponer de información simbólica, la mayoría de los sistemas objetivos podrán ejecutar ejecutable sin dicha información.

Operaciones como desensamblar, construir el marco de la pila y mostrar los registros, es posible que trabajen sin información de símbolos. En algunos casos como al desensamblar, GDB usaría información simbólica para presentar direcciones relativas a símbolos mejor que como números en formato plano o sin formato, pero trabajará igualmente.

Hosts se refiere a la máquina que ejecuta GDB. Target se refiere al sistema donde el programa que esta siendo depurado se ejecuta. En la mayoría de los casos son la misma máquina.

La información necesaria para manejar el sistema del target es dependiente de la arquitectura del target. Ejemplos son el formato del marco de la pila, el juego de instrucciones, las instrucciones de breakpoint, los registros, y como crear o deshacer el marco de la pila en la llamada de una función.

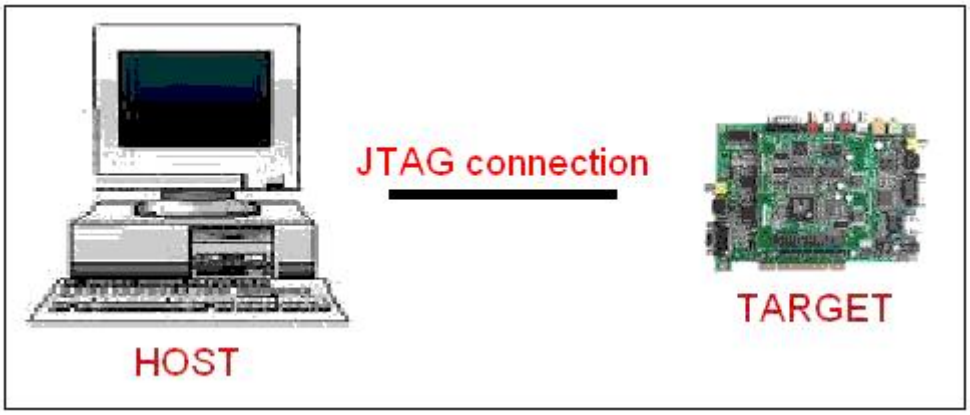


Figura 4: Diagrama Host-Target

3.1.3 Estructura del código fuente de GDB

El directorio fuente del GDB tiene mayoritariamente una estructura plana, hay únicamente unos cuantos directorios. El nombre de un archivo usualmente sólo da una idea o sugerencia sobre lo que hace; por ejemplo 'stabsread.c' lee stabs, 'dwarfread.c' lee DWARF, etc.

Los ficheros que están relacionados por tareas comunes tienen nombres que comparten subcadenas comunes. Por ejemplo, todos los archivos '*-thread.c' hacen referencia a los hilos encargados de la depuración en varias plataformas, los archivos '*read.c' se relacionados con la lectura de varios tipos de símbolos y ficheros objetos, los archivos 'inf*.c' se relacionan con el control directo del programa a depurar.

Hay varias docenas de estos archivos de la familia '*-tdep.c'. Cada uno de estos archivos 'tdep' (target dependent code) implementa un soporte para depurar una arquitectura específica (sparc, mips, CoolFlux). Normalmente, sólo uno de estos será utilizado en una configuración específica del GDB.

Similarmente, hay muchos archivos '*-nat.c' cada uno de los cuales proporciona depuración en modo nativo en un sistema específico (por ejemplo 'sparc-linux-nat.c' da soporte de depuración nativa en maquinas de tipo sparc que ejecutan el kernel de Linux.

3.2 Symbol Side

3.2.1 Lectura de símbolos

GDB lee símbolos desde ficheros de símbolos. El fichero de símbolos, normalmente es el archivo que contiene el programa con el que GDB esta depurando. GDB puede ser utilizado para leer un archivo de símbolos diferente (con el comando 'symbol- file' y también puede leer mas ficheros de símbolos mediante el comando 'add-file' y el comando 'load' , o mientras lee símbolos desde librerías compartidas.

Los ficheros de símbolos son inicialmente abiertos por un código que se encuentra en el archivo 'symfile.c' usando la librería BFD. BFD identifica el tipo de archivo del fichero examinando su cabecera. Find_sym_fns entonces usa esta identificación para localizar un juego de funciones de lectura de símbolos.

Los módulos lectores de símbolos se identifican en GDB mediante una llamada a add_symtab_fns durante la inicialización de sus módulos. El argumento para añadir add_symtab_fns es una estructura sym_fns la cual contiene el nombre del formato de los símbolos, la longitud del prefijo y los punteros a cuatro funciones. Estas funciones son llamadas en varios momentos para procesar ficheros de símbolos cuya identificación coincide con el prefijo especificado.

Las funciones de cada módulo son:

- **xyz_symfile_init** (struct sym_fns *sf)

  Esta función es llamada desde el symbol_file_add cuando nosotros intentamos leer un nuevo archivo de símbolos. Esta función debería borrar cualquier estado interno y preparar el sistema para leer un nuevo archivo de símbolos. Tener en cuenta que el archivo de símbolos que estamos intentando leer podría ser uno nuevo, o podría ser un archivo secundario de símbolos el cuál esta siendo añadido a una tabla de símbolos existente.

  El argumento de xyz_symfile_init es una nueva estructura sym_fns cuyo campo bfd contiene el BFD correspondiente para la lectura de los símbolos del archivo que estamos intentando leer.

  xyz_symfile_init no devuelve nada, pero puede producir un error si detecta algún problema.

- **xyz_new_init** ()

  Esta función es llamada desde el symbol_file_add cuando descartamos la tabla actual de símbolos. Esta función necesita solamente cambiar el estado interno del módulo de lectura de símbolos, de esta manera la tabla de símbolos visible al resto del GDB será descartada. No tiene argumentos y no devuelve nada. Podría ser llamada después de xyz_symfile_init para leer una nueva tabla de símbolos.

- **xyz_symfile_read**(struct sym_fns *sf, CORE_ADDR addr, int mainline)

  Esta función es llamada desde el symbol_file_add para convertir la lectura de símbolos en un conjunto de tablas llamadas psymtabs o symtabs.

  sf apunta a una estructura symb_fns originalmente pasada por xyz_symb_init.

  Addr es el offset entre la dirección de comienzo del archivo especificado y su verdadero valor en memoria.

  Mainline es 1 si la tabla principal de símbolos esta siendo leída o 0 si lo que se esta leyendo es una tabla secundaria de símbolos.

- **xyz_psymtab_to_symtab** (struct partial_symtab *pst)

  Esta función es llamada desde psymtab_to_symtab si el psymtab no ha sido leído todavía. El argumento es la psymtab que deseamos convertir o transformar en una symtab. Antes de ejecutar el return, pst->readin debería valer 1 y pst->symtab debería contener un puntero a la nueva symtab, o 0 si no había símbolos en el archivo de símbolos.

3.2.2 Tablas parciales de símbolos

GDB tiene tres tipos de tablas de símbolos:

- Tablas de símbolos “full” (symtabs). Estas contienen la información principal sobre símbolos y direcciones.

- Tablas parciales de símbolos (psymtabs). Estas contienen suficiente información para saber cuando leer la parte correspondiente de la tabla de símbolos “full”.

- Tabla de símbolos mínimos (msymtabs). Estas contienen información deducidas de símbolos no depurados.

  Una psymtab es conseguida haciendo una pasada muy rápida sobre la información de depuración de un ejecutable. Pequeños pedazos de información son extraídas, (suficiente para identificar que parte de la tabla de símbolos necesitará ser releída y procesada posteriormente) cuando el usuario necesita la información. La velocidad de esta pasada causa que GDB arranque muy deprisa. Mas tarde, cuando una lectura detallada sea necesaria, ocurre en pequeñas pedazos, en varios momentos y el retraso es prácticamente invisible para el usuario.

  La psymtab también contiene el rango de direcciones de instrucciones que symtab representaría.

  La única razón de que exista una psymtab es para que la symtab sea leída en el momento correcto. Cualquier símbolo puede ser suprimido de una psymtab, cuando eso ocurra, no aparecerá en ella. Desde las tablas parciales de símbolos no se tiene una idea del ámbito, no puedes poner símbolos locales en ellas, de ninguna forma. Psymtabs tampoco proporcionan información acerca del tipo de símbolos.

3.2.3 Formato de fichero objeto

El formato ELF viene con el System V Release 4 (SVR4) Unix. El lector ELF básico esta en ‘elfread.c’.

La especificación completa se puede encontrar en la referencia bibliográfica número 6

3.2.4 Formato de información de depuración

DWARF 2 es una versión mejorada pero incompatible de DWARF 1. El lector del DWARF 2 se encuentra en ‘dwarf2read.c’.

Principios básicos

Toda la información en DWARF2 esta almacenada en binario, excepto por las cadenas de texto, por supuesto. La dificultad del DWARF2 consiste en comprender como trabaja y no en analizar sintácticamente extraños formatos de texto. Cuando un binario es compilado con DWARF2 la información de depuración generada, es almacenada en secciones del binario ELF. Las dos secciones principales que necesitaras:

-.debug_info

-.debug_abbrev

Estas dos secciones están estrechamente unidas y ambas están necesitan conseguir información del DWARF2.

Otro tipo de información esta disponible en el formato del DWARF2, tales como números de líneas, direcciones de funciones y mucho más, pero necesitaras utilizar otras secciones:

.debug_line

.debug_aranges

.debug_pubnames

.debug_frame

Organización

En esta sección tratare de explicar como los datos son organizados en las secciones utilizadas por el formato del DWARF2.

Info y abbrev son usados en conjunto esto quiere decir que es necesario comprender como funcionan juntas. Desafortunadamente es muy complicado explicar cada sección por separado, por ello trataré de explicar ambas al mismo tiempo. Esta es la organización principal de la sección de info.

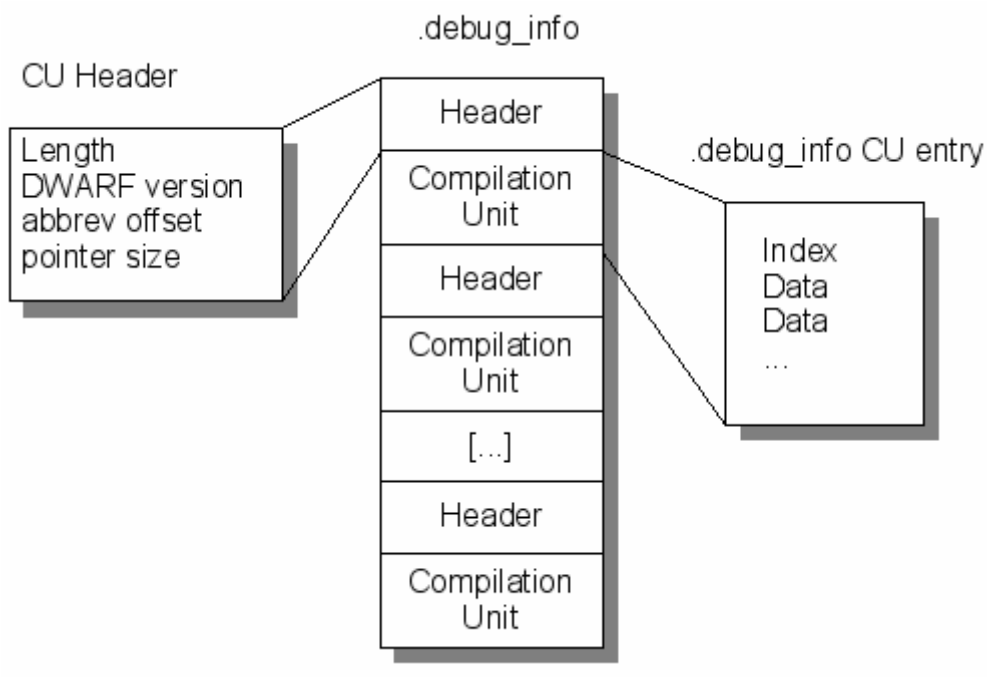


Figura 5: Esquema de la sección .debug_info

No hay una cabecera principal, directamente te encontraras una cabecera que se refiere a la primera unidad de compilación (CU).

¿Qué es una CU? Es una parte de la información DWARF que sólo hace referencia a un objeto o en la mayoría de los casos a un archivo compilado. La mayor parte del tiempo encontraras en una CU el siguiente tipo de información:

- Nombre del archivo
- Ruta donde fue compilado
- El nombre del compilador
- Tipos definidos en cabeceras incluidas
- Tipos básicos usados en el archivo
- Funciones y su tipo de valor de retorno
- Nombre de parámetros y tipos

Como se puede ver en la figura la sección info es una lista de cabeceras CU. Veamos un ejemplo de esta cabecera:

- Longitud del CU: 32 bits
- Versión del DWARF: 16 bits
- abbrev offset: 32 bits
- Tamaño de los punteros: 8 bits

Esto suma un total de 11 bytes. La longitud nos permitirá saber como de largo será la CU sin contar los 11 bytes de la cabecera.

El abbrev offset nos permitirá saber donde encontrar la correspondiente entrada dentro de la tabla .debug_abbrev. Solamente hay que añadir este offset a la dirección base de la sección abrev. Por último el tamaño con el que se va recorrer la tabla de símbolos nos permitirá saber el tamaño de los punteros.

Veamos un ejemplo de la sección abbrev:

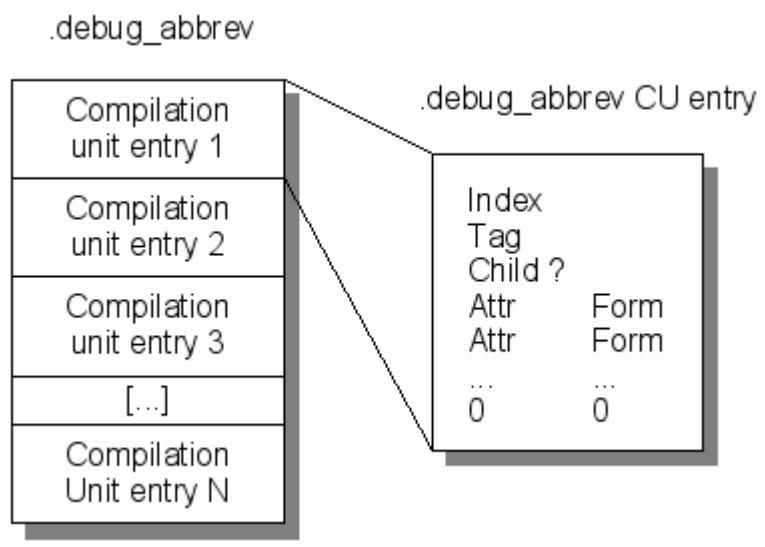


Figura 6: Esquema de la sección .debug_abbrev

La primera cosa que se debería leer en ambas secciones info y abrev. es el índice. Después viene otro *tag* (Figura 6), este nos dirá que es lo que representa la información que viene a partir de este punto, esto se corresponde con DW_TAG_* que se puede encontrar en la documentación oficial. Después viene un flag que indica si dicha sección tiene hijo o no.

En DWARF2 la información crea un árbol en formato binario. La organización en árbol sería algo parecido a esto:

| función

| --> tipo

| --> parámetro

| -----> tipo

| --> parámetro

| -----> tipo

En este ejemplo, la primera entrada indica que a partir de ahora se obtine información de una función con el flag de hijo con valor 1. La segunda entrada tendrá el flag del hijo puesto a 0 y será el tipo de retorno a la función. La tercera tendrá el flag del hijo puesto con 1 y dará el nombre del parámetro. El cuarto tendrá el flag del hijo puesto a 0 y te dará el tipo del parámetro.

Después de este flag siempre aparecerá una pareja de: Attr y Form (Ver figura 6).

El primer Attr te dará información sobre lo que estas intentando leer el nombre, el valor, etc. DWARF2 define DW_AT_* para indicar que el objeto leído se trata de un Attr. En cuanto a Form será muy útil para ser capaces de saber como de larga deberá de ser la lectura en el actual CU. Form viene definido en DWAR2 mediante DW_FORM_* .

La especificación completa se puede encontrar en la referencia bibliográfica número 5.

3.2.5 Añadir un nuevo lector de símbolos a GDB

Si estas utilizando un formato de fichero objeto existente (a.out,COFF,ELF,etc) hay probablemente poco que hacer.

Si necesitas añadir un nuevo formato de fichero objeto, debes añadirlo primero a BFD.

Entonces debes arreglar el código BFD para dar acceso a los símbolos depurados. Generalmente GDB tendrá que llamar a las rutinas de selección de BFD y a otras rutinas internas de BFD para localizar la información de depuración. Tanto como sea posible GDB no debería depender en las estructuras internas de BDF.

Para algunos sistemas (por ejemplo COFF), hay un vector especial de transferencia para llamar a las rutinas de selección, desde el que las estructuras de datos externos pueden tener diferentes tamaños y estructuras en función de las plataformas. En caso

de existir rutinas especializadas para un determinado formato pueden ser llamadas directamente. Esta interfaz debería ser descrita en un fichero 'bfd/libxyz.h' el cual esta incluido en GDB.

3.2.6 Errores detectados

Normalmente no es necesario, retocar nada del Symbol Side, sin embargo para este proyecto fue necesario modificar varios archivos, pero principalmente 'dwarf2read.c', ya que la información de depuración que generaba el compilador, no era 100% estándar. A continuación se muestran algunos ejemplos de errores subsanados:

Nombre de las funciones

Los nombres de las funciones tienen un espacio entre el nombre y su definición. Ello estaba provocando errores de lectura. Este fallo fue arreglado añadiendo algún código en 'dwarf2read', para saltar este tipo de información.

Ejemplo:

```
0000:b140 2d d5 05 52 61 64 69 6f 4d 61 63 5f 53 65 74 50 -Õ.RadioMac_SetP
0000:b150 61 72 61 6d 65 74 65 72 73 20 5f 5f 73 69 6e 74 arameters __sint
0000:b160 5f 52 61 64 69 6f 4d 61 63 5f 53 65 74 50 61 72 _RadioMac_SetPar
0000:b170 61 6d 65 74 65 72 73 5f 5f 5f 50 5f 5f 75 69 6e ameters___P__uin
0000:b180 74 00 00 01 b4 00 01 f5 00 00 0f 18 0d 00 01 b4 t...´..õ.......´
```

Direcciones de las variables

Cada una de las cuatro memorias, que existen tiene de 64K palabras (FFFFh). Para saber cual es la memoria en la que queremos leer/escribir, usamos el siguiente sistema:

PMEM: 0x000000 → 0x00FFFF

IOMEM: 0x010000 → 0x01FFFF

XMEM: 0x020000 → 0x02FFFF

XYMEM: 0x030000 → 0x03FFFF

YMEM: 0x040000 → 0x04FFFF

Además las direcciones de variables que encontramos en la información DWARF, vienen codificadas en LEB128 (Little Endian Base 128).

En el caso del CoolFlux DSP, tienes que restar 0x64. Otro problema es el número de bytes que tenemos que usar para representar direcciones porque pueden cambiar entre 2 y 3 bytes. Esto causa problemas en bastantes ocasiones.

Algoritmo para decodificar signed LEB128

```
result = 0;
shift = 0;
```

```
size = no. of bits in signed integer;
while(true)
{
      byte = next byte in input;
      result |= (low order 7 bits of byte << shift);
      shift += 7;
/* sign bit of byte is 2nd high order bit (0x40) */
      if (high order bit of byte == 0)
        break;
}
if ((shift < size) && (sign bit of byte is set))
/* sign extend */
   result |= - (1 << shift);
```

Algoritmo para decodificar unsigned LEB128

```
result = 0;
shift = 0;
while(true)
{
byte = next byte in input;
result |= (low order 7 bits of byte << shift);
if (high order bit of byte == 0)
break;
shift += 7;
}
```

Información en el Debug_LOC

Una lista de ubicación viene indicada por un atributo de localización cuyo valor es representado por un offset, desde el principio de la sección .debug_loc al primer byte de la lista en cuestión.

Cada entrada en la lista de ubicación se compone de:

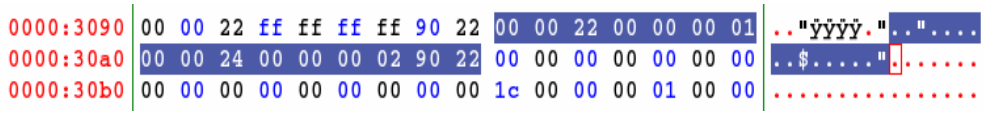
- Una dirección de comienzo. Esta dirección es relativa a la dirección base de la unidad de compilación referenciada en esta lista de ubicación. Marca el principio del rango de direcciones sobre las cuales la localización es válida.

- Una dirección final, otra vez relativa a la dirección base de la unidad de compilación. Marca la primera dirección pasado el final del rango de direcciones sobre el cual la lista de ubicación es valida

- Una expresión de ubicación describiendo la localización del objeto sobre el rango especificado por las direcciones de comienzo y final.

Cada entrada de la lista de ubicación se compone de dos direcciones relativas seguidas por dos bytes de longitud, seguidas por un bloque de bytes continuos. La longitud especifica el número de bytes en el bloque que deben ser leídos.

Sin embargo, para nuestro caso particular, nuestra sección .debug_loc es distinta y sigue el siguiente formato:



Comienzo	– Pipeline Stage	– Final	– Pipeline Stage	– Location expression
00 00 22	00 00 00 01	00 00 24	00 00 00 02	90 22

Los segundos cuatro bytes no parecen que sean usados para indicar la longitud (deberían de ser tan solo dos bytes). Por lo tanto, las expresiones de localización no son correctas y es necesario modificar el código fuente para corregirlas.

Tamaño de los punteros

La mayoría de los punteros son de 3 bytes de longitud, pero hay algunos que sólo tienen una longitud de 2 bytes. Parece que esta diferencia ocurre porque los registros XPTR y YPTR solo son de 2 bytes de longitud, mientras que el ancho de palabra son 3 bytes. Para arreglar esto, nuestros registros XPTR y YPTR están representados internamente como 3 bytes, así podemos usar al mismo tiempo el mismo tamaño de 3 bytes para todos los punteros.

Atributos no estándar en DWARF2

La aparición de estos TAGS se debe a razones internas del depurador de la empresa TARGET. Simplemente es necesario omitirlas cuando las leemos porque GDB no parece necesitar este tipo de información. Esta parte de la documentación es omitida porque TARGET no nos concedió permiso para distribuir su documentación.

Problemas con los parámetros formales

Algunas veces no puedes conseguir información sobre ellos. Este problema ha sido comunicado al TARGET y esperamos será corregido en la próxima versión de su compilador. GDB esta totalmente preparado para dar soporte a las nuevas versiones que solucionen este error.

Información de línea incompleta

Hay muchas líneas en código ensamblador que no se corresponden con ninguna instrucción del código fuente. Por esta razón, una sesión de depuración del programa podría ser bastante difícil algunas veces. Para arreglar esto, mientras GDB esta leyendo la información de la línea, si GDB encuentra una línea sin correspondencia, trataremos de asignar el valor de la línea inmediatamente anterior.

Offset

Hay dos tipos de referencias. El primero es un offset relativo al principio de la unidad de compilación en la cual la referencia se debe referir a una entrada que este dentro de la misma unidad de compilación. El segundo tipo de referencia es una dirección de

cualquier entrada de información depurada con el mismo ejecutable u objeto compartido; debe referirse a una entrada en una unidad de compilación diferente de la unidad que contiene la referencia.

Nuestro problema esta en el primero de los tipos. Los archivos ejecutables generados por el compilador de Target generan offsets absolutos y no relativos. Específicamente los problemas aparecen con el tipo de información DW_ FORM_ ref4. Probablemente este problema será resuelto en la próxima versión del compilador.

3.3 Target Side

3.3.1 Definición de la arquitectura del Target

La arquitectura del Target define que tipos de lenguaje máquina pueden trabajar con GDB, y como trabaja GDB con ellos.

La arquitectura del Target se implementa como una estructura en C del tipo gdbarch*. En dicha estructura hay varios punteros a macros que deben redefinirse para adaptarlos a la nueva máquina.

3.3.2 Inicializando una nueva arquitectura

Cada gdbarch esta asociado con una sola arquitectura BFD o un solo modelo de binario, por medio de una constante bfd_arch_*arch*. El gdbarch se registra mediante una llamada a register_gdbarch_init, normalmente desde la rutina _initialize_*filename*, que será automáticamente llamada durante el arranque de GDB. Los argumentos son una arquitectura BFD y una función de inicialización.

La función de inicialización tiene el siguiente tipo:

- static struct gdbarch ***arch_gdbarch_init*** (struct gdbarch_info info,struct gdbarch_list *arches)

El argumento info contiene los parámetros usados para seleccionar la arquitectura correcta, y arches es una lista de argumentos los cuales, ya han sido creados con el mismo valor de bfd_*arch.*

La función de inicialización debería asegurarse de que la estructura info es aceptable, y devolver NULL si no lo fuera. Después, debería buscar un arches que se corresponda con info y devolverlo. Por último, si no hubiera ninguna correspondencia se debería de crear una nueva arquitectura basada en info y devolverla.

Solo la información en info, debería ser usada para elegir una nueva arquitectura.

3.3.3 Registros y memoria

GDB asume que la maquina incluye un banco de registros y un bloque de memoria. Cada registro puede tener un tamaño diferente.

GDB no tiene ninguna manera de conocer, como el compilador nombrará a los registros, sin embargo, es crítico saber con precisión como lo hace. La única manera para hacer este trabajo es conseguir información acerca de cómo el compilador

asigna estos nombres y reflejarlo en la estructura REGISTER_NAME y en los macros correspondientes.

Mencionar que, GDB puede manejar arquitecturas big-endian, little-endian, y bi-endian.

3.3.4 Punteros y direcciones

En casi todas las arquitecturas de 32-bits, la representación de un puntero es indistinguible de la representación de algunos números, cuyo valor es la dirección de un objeto. En tales maquinas, las palabras “puntero” y “dirección” se pueden intercambiar. Sin embargo, arquitecturas con tamaños más pequeños de palabra, a menudo no tienen un tamaño suficiente para todo el espacio de direcciones, esto significa que se podría elegir una representación diferente para los punteros, la cual permitiera usar un espacio de direcciones más grande.

Por ejemplo, Renesas D10V es un procesador de 16-bits VLIW cuyas instrucciones son de 32 bits de longitud. Si D10V usara una representación normal (byte addresses) para referir las localizaciones del código, entonces el procesador solo sería capaz de direccional 64Kb de instrucciones. Sin embargo, puesto que las instrucciones pueden ser alineadas en paquetes de 4 bytes, los 2 bytes más pequeños de cualquier dirección de una instrucción válida son siempre cero, desperdiciando 2 bits. Para remediar esto, D10V usa direcciones de palabra y direccionamiento por bytes en los que la dirección se corresponde con los dos bits de la derecha. Como consecuencia, D10V puede usar palabras de 16-bit para direccional 256Kb.

Sin embargo, esto quiere decir que los punteros al código y a los datos tienen diferentes representaciones en D10V. La palabra de 16-bits 0xC020 se refiere a la dirección del byte 0xC020 cuando se usa como una dirección a un dato, pero también se refiere al byte 0x30080 cuando se usa como una dirección de código.

Para intentar solucionar este problema, GDB intenta distinguir entre direcciones, y punteros. GDB incluye una función para convertir los punteros en una dirección y viceversa. Esta función será distinta según la arquitectura.

Aquí se muestran las funciones para convertir punteros y direcciones:

- CORE_ADDR **extract_typed_address** (void **buf*, struct type **type)*

Trata los bytes de buf como un puntero o referencia de un tipo type, y devuelve la dirección que representa, de una manera apropiada para la arquitectura actual. Esto produce una dirección que GDB puede usar para leer la memoria del Target, desensamblar, etc.

Si type no es un puntero o referencia a un tipo, entonces esta función generará una señal de error.

- CORE_ADDR **store_typed_address** (void *buf, struct type *type, CORE_ADDR addr)

Guarda la dirección addr en buf, en el formato correcto para un puntero del tipo type en la arquitectura actual.

Si type no es un puntero o referencia a un tipo, entonces esta función generará una señal de error.

- CORE_ADDR **value_as_address** (struct value *val*)

Asumiendo que val es un puntero, delvoverá la dirección que representa, en la forma apropiada para la actual arquitectura.

Esta función, funciona con valores y con punteros. Para los punteros, se realiza una conversión específica como se describió anteriormente para extract_typed_address.

- CORE_ADDR **value_from_pointer** (struct type *type, CORE_ADDR addr)

Crea y devuelve un valor que representa un puntero de tipo type, a la dirección addr, en el formato propicio. Esta función, funciona con valores y con punteros. Para los punteros, se realiza una conversión específica como se describió anteriormente para store_typed_address.

A continuación se muestran algunos macros, los cuales indican las relaciones entre punteros y direcciones. Estos marcos tienen definiciones por defecto, apropiadas para arquitecturas en las cuales todos los punteros son direcciones sin signo.

- CORE_ADDR **POINTER_TO_ADDRESS** *(*struct type *type, char *buf*)*

Asume que buf contiene un puntero de tipo type, en el formato apropiado para la arquitectura actual. Devuelve la dirección, a la cual el puntero se refiere.

- void **ADDRESS_TO_POINTER** (struct type *type, char *buf, CORE_ADDR addr)

Guarda en buf un puntero del tipo type, representando la dirección addr, en el formato apropiado, para la arquitectura actual.

3.3.5 Clases de direcciones

A veces, la información sobre diferentes tipos de direcciones está disponible a través, de la información de depuración Por ejemplo, algunos entornos de programación definen direcciones de varios tamaños. Si la información de depuración distingue estas clases de direcciones gracias a la información sobre el tamaño (por ejemplo DW_AT_byte_size en DWARF 2) o mediante un atributo de clase específico (por ejemplo, DW_AT_address_class en DWARF 2), los siguientes macros deberían ser definidos, para tratar de resolver la ambigüedad de estos tipos en GDB.

- int **ADDRESS_CLASS_TYPE_FLAGS** (int byte_size, int dwarf2_addr_class)

Devuelve un flag necesario para construir un puntero del tipo correspondiente al tamaño byte_size y cuya clase de dirección es *dwarf2_addr_class.* Esta función es normalmente llamada en el lector de símbolos. Ver 'dwarf2read.c'

- char ***ADDRESS_CLASS_TYPE_FLAGS_TO_NAME** (int type_flags*)*

Dándole un flag, devuelve el nombre de la clase correspondiente.

- Int **ADDRESS_CLASS_NAME_to_TYPE_FLAGS** (int name, int *var{type_flags_ptr})

Dándole un nombre de una clase, asigna el int referenciado por type_flags_ptr al tipo correspondiente para la dirección de la clase.

Puesto que es infrecuente encontrar distintas clases de direcciones, estos macros no se definen por defecto. Sin embargo, si se definen, automáticamente son detectados por GDB.

Considerar la arquitectura, en la cual, las direcciones son de 32-bits, pero también se soportan direcciones de 16-bits. Además, suponer que la información DWARF 2 para esta arquitectura solo usa el valor 2 en DW_AT_byte_size, para indicar el uso de un puntero “pequeño”. Las siguientes funciones podrían ser definidas para implementar las clases de memoria.

```
somearch_address_class_type_flags (int byte_size, int dwarf2_addr_class)
{
 if (byte_size == 2)
   return TYPE_FLAG_ADDRESS_CLASS_1;
 else
   return 0;
}

static char *
somearch_address_class_type_flags_to_name (int type_flags)
{
 if (type_flags & TYPE_FLAG_ADDRESS_CLASS_1)
   return "short";
 else
   return NULL;
}

int
somearch_address_class_name_to_type_flags (char *name,int *type_flags_ptr)
{
 if (strcmp (name, "short") == 0)
   {
     *type_flags_ptr = TYPE_FLAG_ADDRESS_CLASS_1;
     return 1;
   }
 else
   return 0;
}
```

La cadena short es usada en GDB, para indicar la presencia de un puntero del tipo “ short”. Así GDB podría mostrar el siguiente comportamiento:

(gdb) ptype short_ptr_var

type = int * @short

Capítulo
4

Desarrollo

4.1 Target Side CoolFlux

CoolFlux fue integrado en GDB en el “Target Side”, porque no es capaz de lanzar GDB por si mismo. Normalmente este tipo de procesadores usan un protocolo definido por GDB para comunicar con el chip. En nuestro caso, nos comunicamos de dos maneras:

- A través de Internet, nosotros mandamos comandos JTAG, que serán recibidos por un driver en el ordenador donde tenemos el DSP conectado.
- A través del puerto paralelo, ya que en remote-CoolFlux.c, se añadió soporte para manejar directamente el driver.

A alto nivel, podemos decir que todo el trabajo de añadir una nueva arquitectura, se realiza por 2 archivos:

4.1.1 Remote-CoolFlux.c

Este archivo controla todas las comunicaciones con el dispositivo, y también todas las operaciones y comandos que podemos manejar, como por ejemplo, colocar breakpoints, hacer un step, CoolFlux_enable_cache, etc. Este archivo también tiene integrado un driver para usar el puerto paralelo en Windows mediante Cygwin.

Al final de este manual, se puede encontrar una plantilla. Solo es necesario completar los “TO DO” y cambiar el nombre CoolFlux por otro distinto.

4.1.2 CoolFlux-tdep.c

Este archivo define la arquitectura del DSP y analiza la información obtenida desde remote_CoolFlux.c. De esta manera este archivo, contiene funciones como el backtrace, pero también define tamaños de datos y la estructura de los registros.

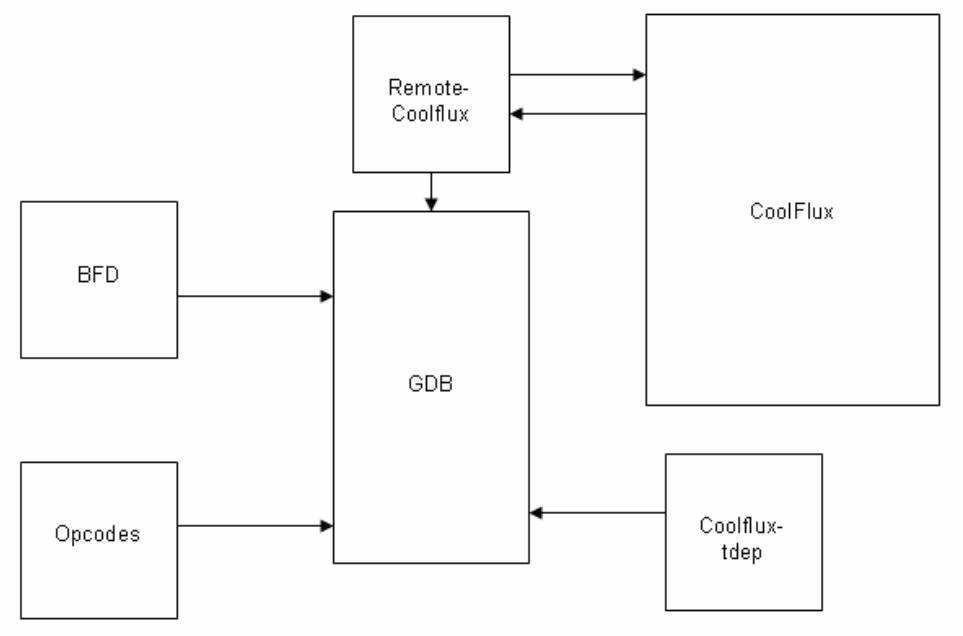


Figura 7: Esquema general de GDB

4.2 CoolFlux-Tdep

GDB usa un número específico de algoritmos de depuración. Frecuentemente no son muy complicados, salvo en casos especiales. Este capítulo describe los algoritmos básicos y menciona algunas de las definiciones específicas del target que se utilizan.

4.2.1 Frames (Marcos)

Un marco es estructura que GDB usa para guardar un registro los marcos de activación de las funciones.

El modelo del marco de GDB, fue implementado para dar soporte a DWARF CFI (CALL FRAME INFORMATION).

El modelo GDB es aquel en el que puedes encontrar los registros de los marcos haciendo "unwinding" sobre el siguiente marco más joven. Por ejemplo, "get frame register" devuelve un valor de un registro en el marco #1(el siguiente al marco más joven), se implementa llamando al frame_register_unwind del frame #0. Pero entonces la pregunta obvia es como accedes a los registros del marco mas joven.

Para contestar esta pregunta, GDB tiene un marco centinela, el marco -1. Por ejemplo, "Unwinding" desde el marco centinela te da los valores del marco de registros más joven. Si f es un marco centinela, entonces get_frame_type (f) == SENTINEL_FRAME.

4.2.2 Análisis del prólogo

Para producir un backtrace o simulación de un estado anterior y permitir al usuario manipular las variables y argumentos de los marcos más antiguos GDB necesita encontrar la base de direcciones de los marcos más antiguos y descubrir dónde se guardan los registros de los marcos.

Versiones modernas de GCC generan Dwarf Call Frame Information (CFI), el cual describe como encontrar las direcciones base del marco y como encontrar los registros guardados. Pero CFI no esta disponible en este caso, así que como una vuelta atrás GDB utiliza una función CoolFlux_analyze_prologue para encontrar el tamaño del marco y los registros guardados.

CoolFlux_analyza_prologue desensambla el código maquina de la función comenzando desde su punto de entrada y busca instrucciones que añadir al espacio del marco, actualizar el puntero de pila y salvar registros.

Obviamente, esto no puede hacerse minuciosamente en general, aún así es suficiente para ser bastante útil.

Para tratar de solucionar este problema, el código en 'prologue_value.h' y 'prologue_value.c' proveen un marco general de trabajo para escribir analizadores de prólogo que sean mas simples. Cuando analizamos un prólogo utilizando el framework estamos haciendo realmente una interpretación abstracta o una seudo evaluación: ejecutando el código de la función en simulación, pero usando aproximaciones conservativas del valor de estos registros y memoria.

En este caso, hacemos la función analyze_prologue, la cual detecta cambios en el puntero de la pila, y en los registros salvados en la pila.

Por ejemplo, si la función comienza con el comando o instrucción:

Sp+=3

Sabemos exactamente que valor contendrá sp después de ejecutar esta instrucción.

Es frecuente para prólogos guardar registros en la pila Así necesitaremos rastrear los valores a través del marco de la pila, así como en los registros, Así que después de una instrucción como:

*(sp-2)=x0

Sabemos exactamente que el valor x0 estará en la pila, entonces podemos recuperar estos registros haciendo unwind.

Por supuesto este análisis no se puede hacer mas allá de los límites que son razonables. Si queremos ser capaces de decir algo sobre la información de la pila, nosotros solamente haremos una aproximación conservativa; esto significa que podemos ignorar las instrucciones que no sean relevantes para prólogos. Esta perdida de información es aceptable para nuestra aplicación.

Por esta razón solo consideramos analizar la información sobre el puntero de la pila y los registros guardados en ella.

Normalmente el procesador tiene una pequeña zona de código al comienzo de la función llamada prólogo. Después de varias instrucciones el programa encontrará una clase de instrucciones las cuales no deberían estar en el prólogo. Esto indica que hemos acabado el prólogo y podemos finalizar el análisis. Pero desafortunadamente, en CoolFlux DSP nunca sabemos cuando el prólogo acaba, por razones de optimización de código. Para arreglar esto leemos desde el principio de la función hasta la línea actual en la que esta detenido el programa.

Para ver cual es el tamaño del marco de pila hay que chequear el valor del registro del puntero de la pila, si es el valor original del SP mas una constante, entonces esa constante es el tamaño del marco de pila.

Para ver donde hemos salvado previamente los registros del marco, buscamos los valores que tenemos para un valor igual al registro original. Si las llamadas a funciones dan un lugar estándar para salvar un determinado registro entonces podemos buscar allí primero (registro LR).

En este caso, nuestra función sólo necesita unas pocas clases de instrucciones. Eso debería ser suficiente para conseguir un poderoso backtrace.

4.2.3 Manejador de los breakpoints

En general un breakpoint es una localización designada en el programa donde el usuario quiere retomar el control si la ejecución del programa alguna vez alcanza esa localización.

Hay dos maneras principales para implementar breakpoints : hardware breakpoints y software breakpoints.

- Hardware breakpoints están disponibles sólo en algunos chips. La manera de implementar hardware breakpoint es mediante un registro dedicado en el cual se guardaran las direcciones donde queremos que el programa se detenga. Si el PC (Program counter) alguna vez coincide con un valor en un registro breakpoint, la CPU lanza una excepción y se lo comunica a GDB. CoolFlux tiene 4 hardware breakpoins, pero uno de ellos esta fijo en la dirección 7 de PMEM, para emular los software breakpoinsts.

  Puesto que consumen hardware, hardware breakpoints deberían ser limitados en número, cuando el usuario sobrepase dicho número, GDB comenzará colocando software breakpoints que explicamos a continuación.

- Software breakpoints requiere a GDB hacer algo mas de trabajo. La teoría básica es que GDB reemplazará una instrucción del programa con la instrucción de ensamblador “swi 7” que causa un salto a una rutina de interrupción que contiene un hardware breakpoint, al retornar de dicha rutina GDB se parará en PC+1, así que tenemos que escribir en el pc con la dirección correcta. Cuando el usuario decide continuar, GDB restaurará la instrucción original, hará un single step, reinsertará el swi 7 y continuará.

  Dado que literalmente este proceso sobreescribe el programa que esta siendo analizado o probado, el área del programa debe ser escribible, así que esta técnica no trabajara en programas en ROM. También puede distorsionar la conducta de los programas que examina, aunque esa situación es altamente inusual.

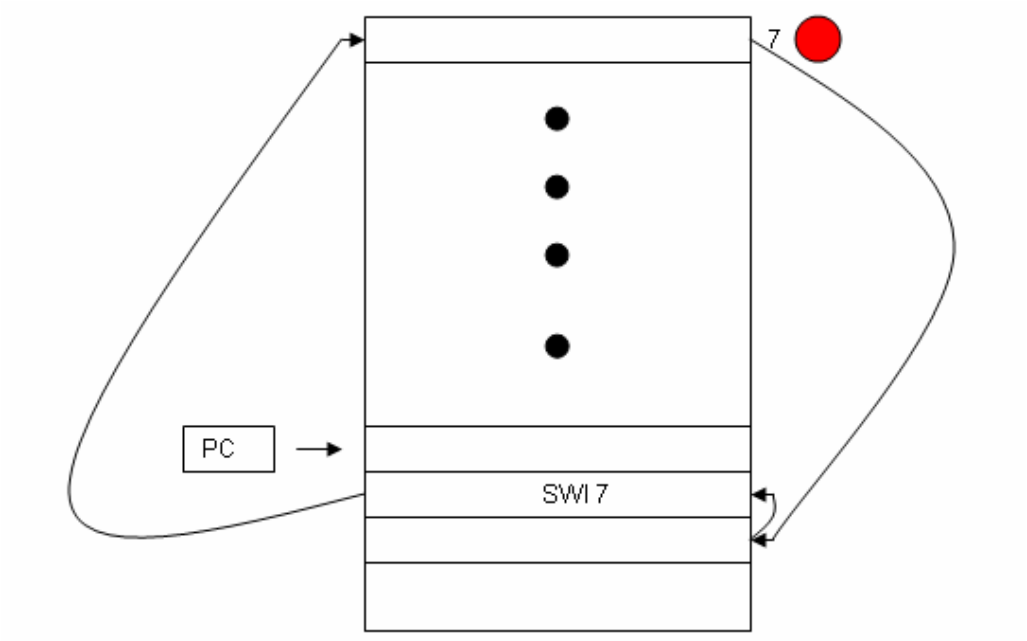


Figura 8: Algoritmo de ejecución de software breakpoints

Hay algunas reglas especiales sobre el uso de los breakpoints:

- Nunca será posible poner un breakpoint en direcciones 0-7
- En algunas instrucciones, como en delay slots no esta permitido el uso de breakpoints
- Cuando se usa un software breakpoint, en ROM, automáticamente será reemplazado por un hardware breakpoint.

La definición básica de un software breakpoint es el macro BREAKPOINT.

El objeto manipulador básico de breakpoint esta en 'breakpoint.c'. De todas formas muchas de las acciones mas interesantes de breakpoint están en 'remote-CoolFlux.c'.

- remote-remove-breakpoint(bp_tgt)
- remote-insert-breakpoint(bp_tgt)
- remote-remove-hw-breakpoint(bp_tgt)
- remote-insert-hw-breakpoint(bp_tgt)

Insertan o quitan un software breakpoint en dirección bp-tgt->placed address. Devuelve cero si tiene éxito, y distinto de cero si falla. La entrada bp-tgt contiene la dirección del breakpoint, si no se incluye es inicializado a cero. Los campos de la estructura bp target info apuntado por bp_tgt son actualizados para contener otra información sobre breakpoint en la salida. El campo placed_adress podría ser actualizado si el breakpoint fue colocado en la dirección relacionada, el campo shadow_contents contiene el contenido real de los bytes donde el breakpoint ha sido insertado. Si leyendo la memoria retornáramos al breakpoint en lugar de a la memoria subyacente; el campo shadow_len es la longitud de la memoria cacheada en shadow_content si existe, y el campo placed_size es opcionalmente fijada y usada por el target, si pudiera ser diferente del campo shadow_len.

Nota: el número máximo de software breakpoint es 200. Sin embargo si necesitas mas hay una constante en 'remote-CoolFlux.c' para cambiar la cantidad.

4.2.4 Codificación de los registros

En DWARF, se realiza un mapeo de los registros de una determinada arquitectura, asignándoles un número. El mapeo debería se debería elegir, intentando maximizar la densidad y debería ser compartido por todos los usuarios de la arquitectura. El SIG recomienda que el mapeo sea definido por un comité de autorización del ABI2. Pero en el caso de CoolFlux el mapeo de los registros fue elegido por la empresa Target. Por algunas razones de funcionalidad interna, los registros del mapeo no son consecutivos, así que es necesario saltarse los registros que no existen.

```
enum gdb_regnum
{
 E_X0_REGNUM=36, E_X1_REGNUM=37, E_Y0_REGNUM=78, E_Y1_REGNUM=79,
 E_A0_REGNUM=0,  E_A1_REGNUM=1,  E_B0_REGNUM=12, E_B1_REGNUM=13,
 E_LE0_REGNUM=14,E_LE1_REGNUM=15,E_LE2_REGNUM=16,E_LE3_REGNUM=17,
 E_LS0_REGNUM=18,E_LS1_REGNUM=19,E_LS2_REGNUM=20,E_LS3_REGNUM=21,
 E_LC0_REGNUM=22,E_LC1_REGNUM=23,E_LC2_REGNUM=24,E_LC3_REGNUM=25,
 E_LP_REGNUM=26, E_XPTR0_REGNUM=38, E_XPTR1_REGNUM=39,E_XPTR2_REGNUM=40,
 E_XPTR3_REGNUM=41,E_XPTR4_REGNUM=42, E_XPTR5_REGNUM=43,E_XPTR6_REGNUM=44,
 E_XPTR7_REGNUM=45,E_XSTEP0_REGNUM=62, E_XSTEP1_REGNUM=63,E_XSTEP2_REGNUM=64,
 E_XSTEP3_REGNUM=65,E_XSTEP4_REGNUM=66, E_XSTEP5_REGNUM=67,E_XSTEP6_REGNUM=68,
 E_XSTEP7_REGNUM=69,E_XMOD0_REGNUM=50, E_XMOD1_REGNUM=51,E_XMOD2_REGNUM=52,
 E_XMOD3_REGNUM=53,E_XMOD4_REGNUM=54, E_XMOD5_REGNUM=55,E_XMOD6_REGNUM=56,
 E_XMOD7_REGNUM=57,E_YPTR0_REGNUM=80, E_YPTR1_REGNUM=81,E_YPTR2_REGNUM=82,
 E_YPTR3_REGNUM=83,E_YSTEP0_REGNUM=88, E_YSTEP1_REGNUM=89,E_YSTEP2_REGNUM=90,
 E_YSTEP3_REGNUM=91,E_YMOD0_REGNUM=84, E_YMOD1_REGNUM=85,E_YMOD2_REGNUM=86,
 E_YMOD3_REGNUM=87, E_SP_REGNUM=34, E_SR_REGNUM=35, E_ISR_REGNUM=30,
 E_LR_REGNUM=32,    E_ILR_REGNUM=28, E_PC_REGNUM=33, E_NUM_REGS=92
};

 static char *register_names[] = {
    "a0", "a1",0,0,0,0,0,0,0,0,0,"b0", "b1","le0","le1","le2","le3","ls0","ls1","ls2","ls3","lc0","lc1","lc2","lc3","lp"
,0,"ilr",0,"isr",0, "lr","pc", "sp", "sr", "x0", "x1","xptr0","xptr1","xptr2","xptr3","xptr4","xptr5","xptr6","xptr7",0,0
,0,0,"xmod0","xmod1","xmod2","xmod3", "xmod4","xmod5","xmod6","xmod7",0,0,0,0,"xstep0","xstep1","xstep2
","xstep3","xstep4","xstep5","xstep6","xstep7",0,0,0,0,0,0,0,0,"y0", "y1","yptr0","yptr1","yptr2","yptr3","ymod0",
"ymod1","ymod2","ymod3","ystep0","ystep1","ystep2","ystep3"
  };
```

4.3 Remote-CoolFlux

4.3.1 PMEM Cache

La gran cantidad de operaciones de lectura que algunas funciones como analyze_prologue, desensamblar,… pueden suponer un deterioro de la velocidad del sistema. Por esta razón, decidimos implementar una cache, la cual básicamente consiste en un array de tamaño igual a la memoria PMEM y otro array para marcar si la posición está actualizada.

Cuando un programa es cargado en el DSP, este también se copia en la cache, lo que se traduce en que no tendremos penalizaciones al leer de PMEM. Otro caso, es cuando un programa ya esta cargado y la cache no está actualizada, entonces la cache se actualizará cuando GDB lea desde PMEM.

Podemos hacer esto, para incrementar la velocidad en los programas que sabemos que no se modifican a si mismos (prácticamente todos). Podemos activar y desactivar la memoria cache en cualquier momento con los siguientes comandos de GDB:

- CoolFlux-cache-enable
- CoolFlux-cache-disable
- CoolFlux cache-flush
- CoolFlux-cache-status

4.3.2 Load

El comando Load analiza todas las secciones del archivo y las carga en la memoria correspondiente. Esto no es tan fácil como parece, puesto que CoolFlux DSP tiene 4 memorias diferentes, lo que implica generar información no Standard, para ser capaz de verificar que zona de memoria corresponde a cada sección. De esta forma, al final del archivo, podemos ver que hay tres secciones que muestran información de cómo debe ser cargado en CoolFlux.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x033ac6 0x00000000 0x00000000 0x00000 0x0000c RW  0
  LOAD           0x033ac6 0x00000000 0x00000000 0x05d30 0x05d30 R E 0
  LOAD           0x0397f6 0x00000c00 0x00000c00 0x0000f 0x0000f R   0
  LOAD           0x039805 0x00000c0f 0x00000c0f 0x00006 0x00006 RW  0
  LOAD           0x03980b 0x00000c15 0x00000c15 0x02652 0x02652 R   0
  LOAD           0x03be5d 0x00003267 0x00003267 0x0000f 0x0000f RW  0
  LOAD           0x03be6c 0x00003276 0x00003276 0x01caa 0x01caa R   0
  LOAD           0x03db16 0x00004f20 0x00004f20 0x00000 0x0007e RW  0
  LOAD           0x03db16 0x00005400 0x00005400 0x00000 0x006c3 RW  0
  LOAD           0x03db16 0x00006000 0x00006000 0x00000 0x006c0 RW  0
  LOAD           0x03db16 0x00006c00 0x00006c00 0x00000 0x02e47 RW  0
  LOAD           0x03db16 0x00000000 0x00000000 0x00363 0x00363 R   0
  LOAD           0x03de79 0x00000480 0x00000480 0x000d8 0x000d8 R   0
  LOPROC+123456  0x03df51 0x00000000 0x00000000 0x00068 0x00068     0
  LOPROC+123457  0x03dfb9 0x00000000 0x00000000 0x00020 0x00020     0
  LOPROC+123458  0x03dfd9 0x00000000 0x00000000 0x00010 0x00010     0
```

Figura 9: Información obtenida con el comando de UNIX readelf

Las últimas 3 cabeceras son especiales (LOCPROC),y son generadas por el compilador, para aportar información extra. En este caso la primera, nos da información sobre donde debemos guardar los datos, la segunda nos muestra que memorias serán usadas por el programa, y la tercera contiene información acerca de la pila.

```
0003:df50 d0 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 Ð...............
0003:df60 07 00 00 00 02 00 00 00 0c 00 00 00 03 00 00 00 ................
0003:df70 0c 00 00 00 04 00 00 00 0c 00 00 00 05 00 00 00 ................
0003:df80 0c 00 00 00 06 00 00 00 0c 00 00 00 07 00 00 00 ................
0003:df90 0c 00 00 00 08 00 00 00 0c 00 00 00 09 00 00 00 ................
0003:dfa0 0c 00 00 00 0a 00 00 00 0c 00 00 00 0b 00 00 00 ................
0003:dfb0 11 00 00 00 0c 00 00 00 11 00 49 4f 4d 45 4d 00 ..........IOMEM.
0003:dfc0 50 4d 45 4d 00 58 4d 45 4d 00 59 4d 45 4d 00 3c PMEM.XMEM.YMEM.<
0003:dfd0 75 6e 6b 6e 6f 77 6e 3e 00 00 00 00 00 00 00 04 unknown>........
0003:dfe0 00 00 00 00 0c 00 00 00 16                      .........
```

Figura 10: Código hexadecimal perteneciente a un archivo compilado

El primero de los rectángulos negros indica la dirección de comienzo de la pila y el segundo la dirección final. Los rectángulos verdes indican el número de sección y los bytes anteriores indican el tipo (IOMEM, XMEM, YMEM XYMEM, y PMEM).

4.3.3 Driver (Socket + JTAG)

Normalmente, todos los 'targets', usan un protocolo definido por GDB, sin embargo, CoolFlux usa uno distinto, para sus comunicaciones. Hay 2 partes, la primera reside dentro de remote-CoolFlux e intenta traducir los comandos que GDB quiere enviar a comandos JTAG específicos de CoolFlux. Una vez, hecha dicha traducción se envía a través de Internet, por medio de sockets. La segunda parte consiste de una aplicación que recibe los comandos JTAG enviados por GDB y los transmite al CoolFlux por medio del puerto paralelo.

4.3.4 Driver (Parallel Port + JTAG + Cygwin)

El mismo archivo, remote-CoolFlux incluye un driver para el puerto paralelo (Solo versión Windows ya que es la plataforma que usan los desarrolladores). Por consiguiente, también podemos depurar de manera local, eliminando así los retrasos propios que se derivan del uso de sockets. El funcionamiento es similar al del driver descrito anteriormente, pero en esta ocasión, la aplicación servidor, está directamente integrada con el remote-CoolFlux.

Como es lógico, para poder usar un driver de puerto paralelo, sobre Windows, es necesario, poder ejecutar nuestra versión de GDB sobre Windows. Se comprobó, que el código que escribimos, es 99% compatible con Cygwin, lo que nos permite usar GDB sobre Windows.

> Nota: Se detectó un pequeño fallo, al usar GDB sobre Cygwin, al intentar usar Ctrl-C. Parece ser que, la pulsación de dichas teclas, es capturada por Windows y no siempre llega a Cygwin.

4.4 Portando GDB

La mayoría del trabajo, para producir una versión de GDB para una nueva arquitectura, reside en la especificación de la configuración de dicha arquitectura. Supongamos que el nuevo host se llama *xyz*, siendo la traducción de xyz → arch-xvend-xos (arquitectura-vendedor-SO). Por ejemplo, nuestra nueva arquitectura viene definida por 'CoolFlux-nxp-none'. Otro ejemplo sería 'sparc-sun-sunos4'.

Pasos a seguir:

1) En el directorio principal, es necesario editar 'config.sub' y añadir arch,xvend y xos, a la lista de arquitecturas soportadas, vendedores, y SO. Se añadirán cerca del final del archivo. También es necesario añadir un alias de xyz. Es posible probar los cambios ejecutando el siguiente comando:

```
./config.sub xyz y ./config.sub arch-xvend-xos
```

2) Ambos comandos deberían responder con *arch-xvend*-xos y no dar ningún mensaje de error.

3) Ahora sería necesario portar el BFD, como se describe en un capítulo del ‘simbol side’. No obstante, en nuestro caso usa ELF32, el cual ya trae implementado soporte.

4) Para que GDB se configure automáticamente, editar ‘gdb/configure.tgt’, asignándole a gdb_target un valor apropiado (por ejemplo, xyz).

5) Finalmente, necesitaremos especificar y definir los archivos ‘.h’ y ‘.c’ dependientes del target y que son usados para la configuración. También es posible definir los opcodes, si se quisiera hacer el desensamblado (ver Apéndice C).

4.4.1 Añadiendo una nueva plataforma objetivo (TARGET)

Los siguientes archivos añaden un target a GDB de manera genérica:

1) **gdb/config/arch/ttt.mt**

   Contiene un fragmento especifico del Makefile para el target. Especifica que archivos objeto son necesarios para el target ttt, mediante la definición de ‘TDEPFILES=...' y ‘TDEPLIBS=...'. También especifica el archivo de cabecera, que describe ttt, mediante la definición de ‘TM_FILE= tm-ttt.h'.

   Puedes también definir ‘TM_CFLAGS', ‘TM_CLIBS', ‘TM_CDEPS', pero están obsoletos, y podrían ser eliminados en futuras versiones de GDB

2) **gdb/ttt-tdep.c**

   Contiene código muy variado requerido por el target (ver definición de las funciones en ‘Target Side’). En algunas arquitecturas no existe. A veces los macros en ‘tm-ttt.h' pueden llegar a ser muy complicados, por lo que normalmente implementan funciones, y el macro simplemente define una llamada a una función. Esta forma de programación a resultado ser muy útil, ya que hace más fácil de comprender y depurar el código

3) **gdb/arch-tdep.c**

4) **gdb/arch-tdep.h**

   Estos archivos a menudo describen el esquema básico del procesador de la nueva arquitectura(registros, pila, etc…). Estos archivos pueden ser compartidos incluso con otros muchos targets que usen el mismo procesador.

5) **gdb/config/arch/tm-ttt.h**

   Contiene definiciones de macros sobre algunos aspectos del target (registros, marco de pila, instrucciones).

   Los nuevos targets no necesitan este archivo y no deberían crearlo.

6) **gdb/config/arch/tm-arch.h**

   Estos archivos a menudo describen el esquema básico del procesador de la nueva arquitectura(registros, pila, etc…). Estos archivos pueden ser compartidos incluso con otros muchos targets que usen el mismo procesador.

   Los nuevos targets no necesitan este archivo y no deberían crearlo.

Si se esta añadiendo un nuevo sistema operativo para una CPU existente, es necesario añadir un archivo ‘config/tm-os.h' que describe algunos aspectos del sistema operativo(información extra de la tabla de símbolos, instrucciones de breakpoint, etc.). Después escribe un archivo ‘arch/tm-os.h' con las siguiente líneas, #includes ‘tm-arch.h' y ‘config/tm-os.h'.

4.4.2 Añadiendo CoolFlux

A continuación se muestra todo el proceso de adaptación de GDB a CoolFlux y no de manera genérica como en el apartado anterior.

Portando BFD

Lo primero que hay que hacer es decidir que ficheros objeto y ejecutables por nuestro procesador son guardados en el disco. Las opciones posibles son a.out, coff, y elf. Estos son formatos BFD. En nuestro caso elegiremos el elf32.

1) **bfd/configure.in y bfd/configure**

Añadir la declaración del vector xyz del CoolFlux.

2) **bfd/config.bfd**

Añadir la declaración del vector xyz del CoolFlux.

3) **bfd/Makefile.am**

Añadir los nombres de los archivos fuente y sus dependencias. Los nombres de estos archivos son cpu-CoolFlux.c, elf32-CoolFlux.c, y include/elf/CoolFlux.h.

4) **bfd/archures.c**

Añadir la declaración de la arquitectura, y el puntero a la misma.

5) **bfd/targets.c**

Añadir la declaración del vector de la arquitectura y el puntero al mismo.

6) **include/elf/common.h**

Añadir el número mágico de tu arquitectura. Para ello, elegir un número aleatorio. Normalmente cada empresa responsable de la arquitectura con un número no oficial e_machine , debería solicitar un número mágico a registry@caldera.com. Nosotros no lo hicimos.

```
/* NXP CoolFlux */
#define EM_COOLFLUX         0x2999   //UNOFFICIAL
```

7) **bfd/cpu-CoolFlux.c**

Aquí tienes que escribir los detalles de la arquitectura y su nombre.

```
const bfd_arch_info_type bfd_CoolFlux_arch =
{
  32,                       /* bits per word */
  24,                       /* bits per address */
```

```
  8,                          /* bits per byte */
  bfd_arch_CoolFlux,          /* architecture */
  0,                                    /* Only one machine */
  "CoolFlux",                 /* architecture name */
  "CoolFlux",                 /* printable name */
  0,                          /* section align power */
  TRUE,                       /* the default ? */
  bfd_default_compatible,     /* architecture comparison fn */
  bfd_default_scan,    /* string to architecture convert fn */
  0                    /* next in list */
};
```

8) **bfd/elf32-CoolFlux.c**

Este archivo es el más complicado. Aquí tienes que definir como a los operandos en binario se les asignas sus valores. Esto es necesario, para ser capaces de realojar un binario en un espacio de direcciones. La primera parte, define como los valores son recolocados y la segunda parte es un mapeado de los nombres específicos de recolocación de CoolFlux. Para este ejemplo, usaremos la información genérica. Esto no incluye ninguna información de recolocación, pero no es necesaria ya que CoolFlux no soporta memoria dinámica.

9) **include/elf/CoolFlux.h**

En este archivo se definen los nombres específicos que es necesario definir.

10) **/bfd/bfd.h, /bfd/bfd-in2.h and /bfd/bfd-in 3.h**

Es necesario insertar la siguiente definición:

```
bfd_arch_CoolFlux,       /* NXP */
#define bfd_mach_CoolFlux      1
```

11) **/bfd/targetmach.h**

```
#if !defined (SELECT_VECS) || defined
(HAVE_bfd_elf32_CoolFlux_vec){
"CoolFlux-*-*",
&bfd_elf32_CoolFlux_vec
},
#endif
```

12) **Compilar en src/bfd**

Si se modificaron todos los archivos mencionados anteriormente (excepto los dos últimos):

```
autoreconf
./configure --target=CoolFlux-nxp-elf
make
```

Después de esto el archive libbfd.a debería existir.

Añadiendo opcodes generados con CGEN

Ahora, trabajaremos en las instrucciones del procesador. El primer paso es copiar **CoolFlux-asm, CoolFlux-desc, CoolFlux-dis, CoolFlux-ibld. CoolFlux-opc.c.** Estos archivos han sido generados con CGEN, más detalles en el apéndice).

1) **opcodes/configure.in y opcodes/configure**

Añadir la arquitectura objetivo

2) **opcodes/Makefile.am**

Añadir las dependencias para la generación automática de archivos, y para las stamp-CoolFlux. El stamp-CoolFlux se usa para llamar a CGEN.

3) **opcodes/disassemble.c**

Añadir la definición del target y la llamada a la función de desensamblado dentro del la estructura case.

```
#ifdef ARCH_CoolFlux
    case bfd_arch_CoolFlux:
      disassemble = print_insn_CoolFlux;
      break;
#endif
```

4) **include/dis-asm.h**

Declarar la función de desensamblado.

```
extern int print_insn_CoolFlux (bfd_vma, disassemble_info *);
```

5) **/include/opcodes/cgen.h**

Es necesario incrementar el valor de una constante:

```
#define CGEN_MAX_IFMT_OPERANDS 16  //FROM 16 TO 30
```

Lógicamente, este paso no será necesario para la gran mayoría de arquitecturas.

Portando GDB (/gdb/)

Ahora trabajaremos con el GDB. El primer paso es copiar los archivos **CoolFlux-tdep** y **remote-CoolFlux** al directorio /gdb/.

1) **gdb/Makefile.in**

Añadir el nombre del código fuente y las dependencias del target. Los archivos fuente son CoolFlux-tdep.c, y remote-CoolFlux.c.

2) **Making GDB**

El directorio principal, editar ‘config.sub' y añadir CoolFlux-*, de esta manera, los vendedores y el sistema operativo no son importantes.

```
./configure --target=CoolFlux-nxp-elf
make
```

Después de varios minutos GDB debería estar listo.

> Nota: Normalmente con esto bastará, pero en nuestro caso debemos hacer algunas modificaciones en el ‘Symbol Side’. Todas las líneas modificadas del ‘Symbol Side’, están marcadas con la palabra “MODIFY”.

Usando GDB

En el directorio principal, ejecutar:

```
./gdb/gdb <filename>
target CoolFlux <ip>
```

Si no se le suministra ninguna dirección IP, entonces GDB intentará conectar con CoolFlux mediante el puerto paralelo. Por comodidad, tal vez, se podrían incluir estos comandos en gdbinit. De esta manera, no tendrías que poner cada vez la misma dirección IP

4.5 Errores no corregidos y limitaciones

En el momento de escribir este manual, nos encontramos con las siguientes limitaciones. Sin embargo, probablemente algunas de estas limitaciones desaparecerán con nuevas versiones del compilador.

4.5.1 Comando CALL

Este comando esta disponible, aunque no es posible utilizarlo con todas las funciones, debido a que:

1) Sólo puede ser utilizado con funciones cuyos tipos de retorno son void, word, dword, edword (tipos básicos), puesto que en la información de depuración esta información no aparece.

2) Algunas veces, en la información de depuración no hay información sobre los parámetros de las funciones.

Estas limitaciones hacen que en ocasiones no se pueda ejecutar el comando call.

4.5.2 Velocidad de depuración

La depuración a través de Internet, es un poco lenta, pero es suficiente para hacer su trabajo. De todas formas, no recomiendo tener simultáneamente en pantalla mucha información, ya que afecta a la productividad del programa.

Depurar a través de puerto paralelo en Windows, hace que la velocidad prácticamente se doble. Este método parece ser suficiente para depurar sin problemas de ralentización graves. Parece recomendable optimizar el código de los sockets, para mejorar la transferencia a través de Internet.

No debería ser difícil reemplazar el puerto paralelo con el USB para incrementar la velocidad. Sin embargo, por falta de tiempo no se terminó el desarrollo del driver.

4.5.3 Problemas con algunas funciones (por ejemplo set_rounding_mode())

Cuando tratas de usar esta función desde su librería original, algunas veces ocurre que el compilador escribe la ruta y después añade la cadena “%procdir%” en la información de depuración, la cual genera un porque el % no es un carácter válido. Para solucionar este problema se recomienda copiar el código de dicha función de la librería al programa que se desea compilar.

4.5.4 Versión ELF

ELF provee un “OBJECT FILE FRAMEWORK” para dar soporte a múltiples procesadores, múltiples códigos de datos y múltiples clases de maquinas. Para dar apoyo a esta familia de archivos objeto, los bytes iniciales del fichero especifican como interpretar el fichero, independiente del procesador.

Los bytes iniciales de una cabecera ELF corresponden al miembro e_ident.

- EI_MAG0 0 File identification
- EI_MAG1 1 File identification
- EI_MAG2 2 File identification
- EI_MAG3 3 File identification
- EI_CLASS 4 File class
- EI_DATA 5 Data encoding
- EI_VERSION 6 File version
- EI_PAD 7 Start of padding bytes
- EI_NIDENT 16 Size of e_ident[]

El byte e_ident[EI_VERSION] especifica la versión de la cabecera ELF. Actualmente, este valor debe ser EV_CURRENT. El valor 1 significa que el fichero está en su formato original. En caso de haber extensiones se aumentará la versión.

Es necesario después de compilar un programa, modificar el 6º byte del archivo, y poner un 1, porque la empresa Target cambia este valor con cada nueva versión.

4.5.5 Memoria ROM

Algunas veces cuando pones un breakpoint en algunas instrucciones en la memoria ROM, es posible que GDB pierda la conexión. Las razones por lo que esto ocurre, no han sido determinadas. Este bug necesita ser investigado, aunque todo parece indicar que el chip CoolFlux salé del modo de depuración.

> Nota: Es importante ver si esta teoría es cierta, para descartar un fallo de diseño del micro

Capítulo 5

Conclusiones y trabajo futuro

5.1 Resultados

Existe una versión comercial de un depurador para CoolFlux realizado por la empresa Target, que también es la encargada de suministrar el compilador para esta arquitectura. Desde este punto de vista, parece absurdo, realizar un nuevo depurador, sin embargo, el depurador oficial, parecía no agradar del todo y se intentó hacer un depurador que lo mejorase. Partiendo de esta base, se buscaron los siguientes objetivos:

1) Inicialmente se me pidió modificar una versión de GDBProxy, para intentar realizar el proyecto, completé el desarrollo pero debido a que el sistema final era demasiado lento se me solicitó la búsqueda de alternativas.

2) Intentar aprovechar código e ideas del depurador oficial. Este objetivo no pudo ser cumplido, ya que el software era propietario y no se nos suministró el código fuente.

3) Generar un depurador, partiendo de un estándar como GDB, puesto que el tiempo de adaptación a este depurador es nulo para cualquier desarrollador. Este objetivo fue totalmente cubierto, e incluso es posible usar el GDB con interfaces gráficas como DDD.

4) Mejorar las funcionalidades de la aplicación comercial. El depurador realizado superó en funcionalidad al comercial, y además es fácilmente ampliable.

5) Como objetivo secundario, se intentó mejorar la velocidad de los drivers (especialmente vía Internet) que existían para conectar el depurador con CoolFlux. Se portó el código de C++ a C y se optimizó. El resultado es que se redujo bastante el lag en red local. También se encontró una posible mejora en la librería de los JTAG, pero no se pudo compilar porque necesitaba una librería propietaria.

6) Se desarrollo un driver para usar el cable paralelo y otro para el USB. El driver para puerto paralelo es 100% funcional y aumenta la velocidad de depuración sobre la opción anterior. Solo se desarrolló en versión Windows. En cuanto al driver USB, no se llegó a terminar por falta de tiempo.

7) Se intentó portar GDB a Windows, mediante el uso de Cygwin. El resultado es 99% funcional, solo se detecta un error al intentar capturar Ctrl.+C, puesto que Windows debe de capturar la pulsación y esta no llega a Cygwin.

Como se observa los objetivos propuestos fueron prácticamente resueltos en su totalidad.

5.2 Conclusiones

El trabajo de portar GDB a otras arquitecturas, no es una tarea fácil, ya que el entorno es muy complejo. Sin embargo, si ya se está familiarizado con este entorno es mucho más sencillo, y el proceso puede realizarse en un tiempo muy reducido, con un mínimo coste. Portar GDB en lugar de desarrollar un depurador adhoc presenta como ventaja una mayor funcionalidad y flexibilidad. Véase por ejemplo, los comandos de cache introducidos para acelerar el funcionamiento del programa.

Reutilizar esta información a nivel de empresa, podría ahorrar mucho tiempo y dinero. Además de usar un interfaz que todo el mundo conoce. Son tantas las ventajas en este sentido, que incluso la empresa Target Compilers (desarrolladora del compilador para CoolFlux) se intereso por el proyecto, para estudiar la implantación de esta tecnología en sus proyectos.

En cuanto a la programación de drivers, si bien es relativamente “simple” construirlos, no es nada fácil optimizarlos, ya que a veces el sistema operativo no gestiona las conexiones como nos gustaría. Tanto es así, que en determinadas ocasiones fue necesario el uso del osciloscopio para detectar, como de buena era una implementación del driver, o simplemente para ver si los retardos de la conexión eran por la implementación del GDB o por problemas en el driver.

No fue posible optimizar todo el código de los drivers, por culpa de una librería propietaria, aunque se encontró que sobraba un estado, en las conexiones JTAG que podría incrementar en teoría un 33% la velocidad de cualquier conexión (USB,Puerto Paralelo). No obstante, incluso sin esta mejora, nuestro driver mejoró sustancialmente la velocidad del driver del depurador oficial.

Los principales problemas encontrados durante el desarrollo del proyecto, fueron sin duda alguna, los cambios constantes en los objetivos y los cambios de versiones. Durante el desarrollo del proyecto se sufrió la aparición de 4 versiones distintas del compilador, 2 de los drivers y una del GDB. Cada cambio de versión sobre todo del compilador, implicaba una reestructuración profundad del código DWARF, lo que implicaba una revisión profunda del GDB. Sobre todo los primeros cambios fueron tan radicales, que se tuvo que prácticamente reiniciar el proyecto. En condiciones normales, esto no sucedería, sin embargo, los compiladores auto-generados de la empresa Target aún no están 100% finalizados, sobre todo en lo que se refiere a información DWARF.

Fruto de la inestabilidad del código DWARF, las herramientas estándares para analizarlo no funcionaban y tuve que modificar algunas herramientas como DWARFDump para poder examinar el código de depuración. Sin embargo, para poder hacer esto, fueron necesarias largas horas de ingeniería inversa y examinar código en hexadecimal.

Como conclusión, este proyecto no solo es un reto, para una persona inexperta, si no que además, es un proyecto que una vez concluido da pie a la realización de multitud de futuros proyectos, como veremos a continuación.

5.3 Trabajos Futuros

5.3.1 GDB CoolFlux + Eclipse

El primer proyecto que se plantea, es la integración de cualquier nueva versión de GDB con Eclipse. Este proyecto, se estudió y en principio no habría ningún problema en integrar nuestro nuevo GDB con Eclipse, al igual que se hace con la versión normal. Más problemático es el caso de integrar el compilador con Eclipse. Sin embargo, después de consultar varios manuales, no parece difícil hacer un plug-in para comunicar el compilador de Target con el Eclipse.

Superado el problema anterior, tendríamos un versión de Eclipse 100% funcional para nuestro chip CoolFlux. De nuevo, apostando por las tecnologías libres, y por las plataformas, que menos tiempo de adaptación necesitan para el personal, ya que todo el mundo conoce Eclipse y GDB.

5.3.2 GDB CoolFlux Multi-Core

Imaginemos por un momento, que tuviéramos, un DSP pero en vez de con un solo núcleo de CoolFlux con tres núcleos. Esta implementación existe, y por supuesto necesita un depurador. La propuesta, consiste en:

1) Adaptar el driver, para que admita dirigir cadenas JTAG a un núcleo específico.

2) Crear una interfaz, que nos permita agrupar tres instancias de GDB, cada una conectada a un determinado núcleo. Esta interfaz, es fundamentalmente, para permitirnos el envío de comandos a los tres núcleos a la vez. Si no fuera necesaria, el envío simultaneo, nos bastaría con el paso anterior. Básicamente la interfaz contendría 3 ventanas de GDB, cada una conectada a un procesador, y un campo para el envío de comando simultáneos.

5.3.3 Depuración con interrupciones

Durante el desarrollo de GDB, se encontró una manera de superar la limitación de 4 breakpoints que nos permite utilizar la arquitectura. La manera consistía en sacrificar un breakpoint en una posición de memoria usada para lanzar instrucciones software(ver más información en capítulo 4).

Posteriormente se estudió la posibilidad de adaptar dicha técnica, a procesadores, que por motivos de tamaño habían tenido que suprimir ciertas patillas de conexión JTAG para disminuir su tamaño. Como resultado en estos chips solo se pude lanzar un programa y consultar el contenido de la memoria.

Nos hemos dado cuenta que podíamos usar la técnica anteriormente descrita, para llamar a una interrupción software que vuelque el contenido de los registros y de ciertas variables, a la memoria. Así una vez detenido el núcleo, toda la información volcada en memoria, nada nos impide emular el comportamiento real del GDB, pero leyendo todo de la memoria, que es la única operación disponible mediante el JTAG reducido.

El resultado para el programador, sería como usar un depurador normal, y por supuesto mejoraría el desarrollo actual.

Capítulo 6

Bibliografía

6.1 Manuales

[1] GDB internals, http://sourceware.org/gdb/current/onlinedocs/gdbint_toc.html

[2] Documentación entregada a NXP sobre CGEN y Opcodes (Corentin Lecouvey)

[3] CoolFlux DSP Assembly Programmer's Manual, Informe técnico interno de NXP

[4] CoolFlux DSP Programmer's Manual, Informe técnico interno de NXP

[5] DWARF Debugging Standars, http://www.eagercon.com/dwarf/dwarf-2.0.0.pdf

[6] ELF Standars, http://www.x86.org/ftp/manuals/tools/elf.pdf

6.2 Libros

[7] Jonathan B. Rosenberg, "How debuggers work", John Wiley & Sons, 1996, ISBN: 978-0471149668

[8] John R. Levine, "Linkers & loaders", Morgan Kaufmann, 2000, ISBN: 978-1558604964

6.3 Palabras clave

GDB, CoolFlux, CGEN, Opcode, Depurador, DWARF

Apéndice A

Funciones para definir el Target

Esta sección describe, los macros que puedes usar para definir el Target.

ADDR_BITS_REMOVE (addr)

Si la dirección de una instrucción incluye algún bit que no es realmente parte de la dirección, entonces hay que definir este macro para que ponga a cero esos bits en addr. Esto sólo se utiliza para las direcciones de instrucciones, y no en todos los contextos.

Por ejemplo, los dos bits de menor peso del PC en la arquitectura de Hewlett-packard PA 2.0 contienen el nivel de privilegio de la correspondiente instrucción. Puesto que las instrucciones deben agruparse en grupos de cuatro bytes, el procesador deshecha/aparta esos bits para generar la dirección real de la instrucción. ADDR_BITS_REMOVE debe filtrar estos bits con una expresión como ((addr) & ~3).

ADDRESS_CLASS_NAME_TO_TYPE_FLAGS (name, type_flags_ptr)

Si name es un nombre válido para la dirección de la clase, asigna el int referenciado por *type_flags_ptr* a la máscara que representa el nombre y devuelve 1. Si name no es un nombre válido para la dirección de la clase, devuelve 0.

El valor para *type_flags_ptr* debe ser uno de TYPE_FLAG_ADDERSS_CLASS_1, TYPE_FLAG_ADDERSS_CLASS_2*,* o alguna posible combinación de estos valores sumados. Ver la sección de Clases de direcciones.

ADDRESS_CLASS_NAME_TO_TYPE_FLAGS_P ()

Predicado que indica si se ha definido ADDRESS_CLASS_NAME_TO_TYPE_FLAGS.

ADDRESS_CLASS_TYPE_FLAGS (*byte*_size, dwarf2_addr_class)

Dado un puntero de tamaño de byte (según lo descrito en la información de depuración) y el valor de DW_AT_address_class, devuelve los flags usados por GDB para representar esa dirección de clase. El valor devuelto debe ser uno de TYPE_FLAG_ADDRESS_CLASS_1, TYPE_FLAG_ADDRESS_CLASS_2, o alguna

posible combinación de estos valores sumados. Ver la sección de Clases de direcciones.

ADDRESS_CLASS_TYPE_FLAGS_P ()

Predicado que indica si se ha definido el ADDRESS_CLASS_TYPE_FLAGS.

ADDRESS_CLASS_TYPE_FLAGS_TO_NAME (type_flags)

Devuelve el nombre de la dirección de la clase asociada con el flag dado por *type_flags*.

ADDRESS_TO_POINTER (type, buf, addr)

Almacena en *buf* un puntero de tipo *type* que representa la dirección *addr*, en el formato apropiado para la arquitectura actual. Esta macro asume que *type* es un puntero o una referencia a un tipo de C++. Ver la sección Los punteros no son siempre direcciones.

BELIEVE_PCC_PROMOTION

Definir si el compilador pasa un parámetro short o char a int, pero sigue refiriéndose al parámetro en su tipo original.

BITS_BIG_ENDIAN

Un valor de 1 significa que los bits están numerados en un big-endian, 0 significa little-endian.

BREAKPOINT_FROM_PC (pcptr, lenptr)

Utiliza el contador de programa para determinar el contenido y tamaño de una instrucción de breakpoint. Devuelve un puntero a un string de bytes que codifican una instrucción de breakpoint, almacena la longitud del string en **lenptr*, y ajusta el contador de programa (si es necesario) al punto de la posición de memoria donde el breakpoint debe insertarse.

Aunque es común utilizar una instrucción trap en los breakpoints, no es necesario; por ejemplo, la configuración de bits podría ser una instrucción invalida. El breakpoint no debe ser más largo que la instrucción más corta de la arquitectura.

Sustituye todas las demás macros de BREAKPOINT.

MEMORY_INSERT_BREAKPOINT (*bp_tgt*)

MEMORY_REMOVE_BREAKPOINT (bp_tgt)

Inserta o quita los breakpoints almacenados en memoria. Se proporcionan definiciones comunes (default_memory_insert_breakpoint y

default_memory_remove_breakpoint respectivamente) de modo que no sea necesario definirlos para la mayoría de las arquitecturas. Las arquitecturas que pueden necesitar definir MEMORY_INSERT_BREAKPOINT y MEMORY_REMOVE_BREAKPOINT tendrán instrucciones que tienen tamaño no habitual o no están almacenadas de manera convencional.

También puede ser deseable (desde el punto de vista de la eficiencia) definir rutinas de inserción y eliminación de breakpoints propias si BREAKPOINT_FROM_PC necesita leer la memoria del target por alguna razón.

ADJUST_BREAKPOINT_ADDRESS (address)

Dada una dirección en la cual, se desea colocar un breakpoint, devuelve una dirección de breakpoint ajustada de acuerdo con las normas de una determina arquitectura. Normalmente este método no es necesario en la mayoría de arquitecturas.

CANNOT_FETCH_REGISTER (regno)

Es una expresión en C que debería ser distinta de 0 si regno no puede ser fetch desde un proceso inferior. Solo es relevante si FETCH_INFERIOR_REGISTERS no está definido.

CANNOT_STORE_REGISTER (regno)

Es una expresión en C que debería ser distinta de 0 si regno, no debería ser escrito en el Target. Este caso se da a menudo para contadores de programa, y registros especiales. Si no se define, GDB asumirá que todos los registros pueden ser escritos.

int CONVERT_REGISTER_P(regnum)

Return non-zero if register *regnum* can represent data values in a non-standard form. See section Using Different Register and Memory Data Representations.

DECR_PC_AFTER_BREAK

Definir esta constante para indicar la cantidad que decrementará el PC después de que el programa se detenga en un breakpoint. Esto es a menudo el número de bytes en un BREAKPOINT, aunque no lo es siempre. Para la mayoría de Targets este valor será 0.

DISABLE_UNSETTABLE_BREAK (addr)

Si está definido, devolverá 1 si la dirección addr, está en una librería compartida en la cual los breakpoints no pueden ser colocados y deberían ser desactivados.

PRINT_FLOAT_INFO()

Si es definido, entonces el comando ‘info float’ imprimirá información sobre la unidad de punto flotante del procesador.

print_registers_info (gdbarch, frame, regnum, all)

Si es definida, imprimirá el valor del registro regnum para el frame especificado. Si el valor de regnum es -1, se imprimirán todos los registros.

El método por defecto, imprime un registro por línea, y si all es cero omite los registros de punto flotante.

PRINT_VECTOR_INFO ()

Si está definido, entonces el comando 'info vector' llamará a esta función para imprimir información sobre la unidad de vectores del procesador.

DWARF_REG_TO_REGNUM

Convierte los números de registros DWARF en números de registros que maneja GDB. Si no se define, no se ejecutará ninguna conversión.

DWARF2_REG_TO_REGNUM

Igual que la anterior pero con información DWARF 2.

ECOFF_REG_TO_REGNUM

Igual que las anteriores pero esta vez con información ECOFF.

END_OF_TEXT_DEFAULT

Este es una expresión que debería designar el fin de la sección de texto.

frame_align (address)

Hay que definir este macro para ajustar la dirección address, para que cumpla los requisitos de alineamiento para el comienzo de un nuevo marco de pila.

Esta función es usada para asegurarse de que, cuando se crea un nuevo marco, tanto el puntero de pila inicial y la dirección de el valor de retorno son correctamente alineados.

Por defecto, no se produce ningún alineamiento.

FRAME_NUM_ARGS (fi)

Para el marco descrito por fi, devuelve el número de argumentos que están siendo pasados. Si el número de argumentos no es conocido, se devuelve -1.

CORE_ADDR **unwind_pc** (struct frame_info *this_frame)

Devuelve la dirección de la instrucción, en la función que llama a this_frame, en el cual la ejecución continuará después de que this_frame retorne.

La implementación, debe ser agnóstica (trabajar con cualquier marco). Normalmente no es más que:

ULONGEST pc;

frame_unwind_unsigned_register (this_frame, D10V_PC_REGNUM, &pc);

return d10v_make_iaddr (pc);

CORE_ADDR **unwind_sp** (struct frame_info *this_frame)

Devuelve el marco de pila más interno.

FUNCTION_EPILOGUE_SIZE

Para algunos COFF targets, el campo x_sym.x_misc.x_fsize de una función es 0. Para estos targets, debes definir FUNCTION_EPILOGUE_SIZE, para ampliar el tamaño de una función de epílogo.

GDB_MULTI_ARCH

Si está definido y es distinto de cero, se activa el soporte para múltiples arquitecturas en GDB

Este soporte puede ser activado en 2 niveles. En el nivel 1, solo las definiciones para los anteriores marcos no definidos son ajuntadas. En el nivel 2, se definirá una multi-arquitectura, la cual contendrá todos los macros dependientes de todas las arquitecturas.

GET_LONGJMP_TARGET

Para la mayoría de maquinas, este es un parámetro dependiente.

Este macro determina la dirección del PC a la que un longjmp saltará, asumiendo que nosotros hemos parado en un longjmp breakpoint. Lleva como argumento CORE_ADDR* y guarda el valor del PC a través de este puntero.

SYMBOLS_CAN_START_WITH_DOLLAR

Algunos sistemas tienen rutinas cuyos nombres empiezan con $. Cambiando el valor de este macro a algo distinto de 0, le dirá a GDB que analice las rutinas cuyo primer token sea $.

INNER_THAN `(lhs, rhs)`

Devuelve distinto de 0, si la dirección de la pila lhs está más cercana a la cima de la pila que la dirección rhs. Hay que definir este macro como lhs < rhs si la pila del target crece hacía abajo en memoria o si lhs > rhs si la pila crece hacia arriba.

gdbarch_in_function_epilogue_p `(gdbarch, pc)`

Devuelve un valor distinto de cero si el valor del PC está en el epílogo de una función. El epílogo de una función es definido como la parte de una función donde el marco de pila de un función ya ha sido removido del final de la pila.

IN_SOLIB_CALL_TRAMPOLINE `(pc, name)`

Definir este macro para devolver un valor distinto de 0 si el programa esta detenido en el trampolín que conecta con una librería compartida.

IN_SOLIB_RETURN_TRAMPOLINE `(pc, name)`

Definir para evaluar a un valor distinto de 0, si el programa se detiene en el trampolín al retornar desde una librería compartida.

IN_SOLIB_DYNSYM_RESOLVE_CODE `(pc)`

Definir este macro, para evaluar a un valor distinto de 0 si el programa se detiene en el linkado dinámico.

SKIP_SOLIB_RESOLVER `(pc)`

Definir esta función para devolver una dirección en la cual, la ejecución debería continuar para pasar la función de resolución de símbolos del linkado dinámico. Un cero como valor indica que no es importante o necesario colocar un breakpoint para entrar en el linkado dinámico y un solo step será suficiente.

INTEGER_TO_ADDRESS (type, buf)

Hay que definir este macro, cuando la arquitectura necesite manejar conversiones entre punteros y direcciones. Convierte un entero a una dirección de acuerdo con las reglas de la arquitectura actual.

POINTER_TO_ADDRESS (type, buf)

Asume que buf contiene un puntero del tipo type, en el formato apropiado para la arquitectura actual. Devuelve la dirección a la que el puntero hace referencia. Ver la sección de punteros no son siempre direcciones.

REGISTER_CONVERTIBLE (reg)

Devuelve distinto de cero si reg usa diferentes formatos. (raw y virtual)

REGISTER_TO_VALUE(regnum, type, from, to)

Convierte el contenido del registro regnum en un valor del tipo type.

register_reggroup_p (gdbarch, regnum, reggroup)

Devuelve distinto de cero, si el registro regnum es un miembro del grupo de registros regroup.

Por defecto, los registros son agrupados por:

float_reggroup

Cualquier registro, con un nombre válido y un tipo de coma flotante.

vector_reggroup

Cualquier registro con un nombre válido y de tipo vector.

general_reggroup

REGISTER_CONVERT_TO_VIRTUAL(reg, type, from, to)

Convierte el valor del registro reg a su forma virtual.

REGISTER_CONVERT_TO_RAW(type, reg, from, to)

Convierte el valor del registro reg de su forma virtual a su forma normal.

const struct regset ***regset_from_core_section** (struct gdbarch * gdbarch, const char * sect_name, size_t sect_size)

Return the appropriate register set for a core file section with name *sect_name* and size *sect_size*.

SOFTWARE_SINGLE_STEP_P ()

Define este macro como 1 si el Target no tiene mecanismos hardware de single-step. El macro SOFTWARE_SINGLE_STEP debe ser también definido.

SOFTWARE_SINGLE_STEP(signal, insert_breapoints_p)

Una función que inserta o quita breakpoints en cada destino posible de la siguiente instrucción.

SOFUN_ADDRESS_MAYBE_MISSING

Se puede observar, que cuanto más precisas son las direcciones que tienes en la información de depuración, más tiempo ha gastado el linier en generarlas. Siempre que, haya una manera de que el depurador encuentre las direcciones que necesita, se debería omitir dicha información de la información de depuración, para hacer el linier más rápido.

SOFUN_ADDRESS_MAYBE_MISSING indica que un tipo particular del conjunto de estos trucos está en uso, afectando a las entradas N_SO y N_FUN en las stabs-format debugging information. N_SO marca el principio y el final de las unidades de compilación en segmento de texto. N_FUN marca los comienzos y finales de las funciones.

PC_LOAD_SEGMENT

Si se define, imprime información acerca del segmento cargado por el PC.

PC_REGNUM

Si el PC es almacenado en un registro, entonces hay que definir este macro para que sea el número de ese registro

Solo hace falta definirlo si TARGET_READ_PC y TARGET_WRITE_PC no están definidos.

PARM_BOUNDARY

Si es distinto de 0, redondea los argumentos a tantos bits como su valor indique antes de ponerlos en la pila.

stabs_argument_has_addr (gdbarch, type)

Definir esta función para devolver un valor distinto de cero, si un argumento de una función del tipo type es pasado por referencia en vez de por valor.

PS_REGNUM

Si está definido, este es el número del status register.

push_dummy_call (gdbarch, function, regcache, pc_addr, nargs, args, sp, struct_return, struct_addr)

Define esta función, para colocar las llamadas a funciones en la pila. Además, de colocar nargs, el código debería colocar struct_addr, y la dirección de retorno.

function es un puntero a una estructura value. En arquitecturas que usan descriptores de funciones, este contiene el valor del descriptor de la función.

Devuelve el puntero a la cima de la pila.

CORE_ADDR **push_dummy_code** (gdbarch, sp, funaddr, using_gcc, args, nargs, value_type, real_pc, bp_addr)

Dada una pila, pone la secuencia de instrucciones, gracias a la cual, la función llamada debería retornar.

Asigna bp_addr a la dirección en la cual la instrucción de breakpoint debería ser insertada, real_pc se asigna a la dirección de retorno cuando comience la secuencia de la llamada. Devuelve la dirección del marco de pila más interno.

REGISTER_NAME(i)

Devuelve el nombre del registro i como una cadena. Podría devolver NULL para indicar que el registro I no es válido.

SDB_REG_TO_REGNUM

Definir este macro para convertir los números de registros sdb en registros GDB regnums. Si no está definido, no se realizará ninguna conversión.

SKIP_PERMANENT_BREAKPOINT

Avanza el PC, pasando el breakpoint permanente. GDB normalmente intenta eliminar el breakpoint, hacer un step, y reinsertar el breakpoint. Sin embargo, normalmente los breakpoints permanentes están cableados mediante hardware, y no se pueden quitar, por lo tanto esta estrategia no funcionará. Llamar a SKIP_PERMANENT_BREAKPOINT ajustará el estado del PC, por lo que la ejecución se detendrá justo después del breakpoint.

SKIP_PROLOGUE (pc)

Una expresión en C que devuelve la dirección del código “real”, más allá del prólogo de entrada de la función encontrada en el PC.

SKIP_TRAMPOLINE_CODE (pc)

Si el Target tiene código trampolín el cual, permanece entre la función y la llamada a la misma, entonces es necesario definir este macro para devolver el nuevo PC que está al principio de la función real.

SP_REGNUM

Si el puntero de pila es almacenado en un registro, entonces hay que definir este macro para ser el número de ese registro, o -1 si no hay tal registro.

STAB_REG_TO_REGNUM

Hay que definir este macro para convertir los números de registros encontrados en la información de depuración a GDB regnums. Si no se define, no habrá conversión

STEP_SKIPS_DELAY (*addr*)

Define este macro, para devolver verdadero si la dirección es de una instrucción con delay slot. Si un breakpoint ha sido colocado en una instrucción con breakpoint, GDB ejecutará un single-step sobre dicha instrucción. Actualmente solo está definida para el Mips.

TARGET_READ_PC

TARGET_WRITE_PC (val, pid)

TARGET_READ_SP

TARGET_READ_FP

Estos macros cambian el comportamiento de read_pc, write_pc y read_sp. Para la mayoría de Targets, no es necesario definir estas funciones. GDB llamará a las funciones de lectura y escritura con el argumento REGNUM correspondiente.

Estos macros son muy útiles cuando un Taget almacena uno de esos registros en un lugar difícil de localizar. Por ejemplo, parte en un segmento de un registro y parte en un segmento de un registro ordinario.

TARGET_VIRTUAL_FRAME_POINTER (pc, regp, offsetp)

Devuelve una pareja de (registro,offset) representando, el puntero al marco virtual en uso en la dirección pc. Si los punteros a marcos virtuales no se usan, la definición por defecto simplemente devuelve DEPRECATED_FP_REGNUM, con un offset de 0.

TARGET_HAS_HARDWARE_WATCHPOINTS

Si no es 0, el Target tiene soporte para hardware watchpoint.

TARGET_PRINT_INSN (addr, info)

Esta es la función usada por GDB para imprimir una instrucción en esamblador. Imprime la dirección addr y devuelve el tamaño de la instrucción, en bytes. Si un Target no define su propia función, esta será por defecto una llamada a un accesor para el puntero global deprecated_tm_print_insn. Esta normalmente, apunta a una función en la librería de los opcodes. Info es una estructura (del tipo disassemble_info) definida en 'include/dis-asm.h' usada para pasar información a la rutina de decodificación de instrucciones.

struct **frame_id unwind_dummy_id** (struct frame_info *frame)

Dado un marco, devuelve una estructura frame_id que indentifica una llamada a una función. El valor devuelto debe coincidir con el valor del marco de pila guardado previamente usando SAVE_DUMMY_FRAME_TOS.

VALUE_TO_REGISTER(type, regnum, from, to)

Convierte a un valor del tipo type los contenidos de los registros regnum.

Apéndice B

Plantilla para archivo remote-ARCH.c

Esta plantilla es genérica, e incluye todas las funciones que podrían ser necesarias por la gran mayoría de procesadores, aunque es posible que fuera necesario añadir alguna función puntual. Para usar esta plantilla, es necesario seguir los siguientes pasos:

1) Copiar la plantilla y guardarla en /gdb/ como remote-“ARCH”.c

2) Escribir el código necesario.

3) Cambiar todas las apariciones de la palabra CoolFlux por el correspondiente nombre del procesador.

B.1 Código

```
#include "defs.h"
#include "gdb_string.h"
#include <ctype.h>
#include <fcntl.h>
#include "inferior.h"
#include "bfd.h"
#include "symfile.h"
#include "exceptions.h"
#include "target.h"
#include "gdbcmd.h"
#include "objfiles.h"
#include "remote.h"
#include "regcache.h"
#include <ctype.h>
#include <sys/time.h>
#include <signal.h>
#include "gdbcore.h"
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/errno.h>

#define CoolFlux_hw_watchpoint_limit  0
#define CoolFlux_hw_breakpoint_limit  4
#define breakpoint_limit          200
#define MAX_ADDRESS               0xFFFF

static void build_remote_gdbarch_data (void);
static void remote_prepare_to_store (void);
static void remote_fetch_registers (int regno);
static void remote_resume (ptid_t ptid, int step,
                           enum target_signal siggnal);
static void remote_open (char *name, int from_tty);
static void remote_open_1 (char *, int, struct target_ops *);
static void remote_load (char *args, int from_tty);
static void remote_close (int quitting);
```

```
static void remote_store_registers (int regno);
static void remote_mourn (void);
static void remote_mourn_1 (struct target_ops *);
static ptid_t remote_wait (ptid_t ptid,
                           struct target_waitstatus *status);
static void remote_kill (void);
static void remote_detach (char *args, int from_tty);
static void remote_interrupt (int signo);
static void remote_interrupt_twice (int signo);
static void interrupt_query (void);
static void init_remote_ops (void);
static void remote_stop (void);
static CORE_ADDR remote_address_masked (CORE_ADDR);
static int fromhex (int a);
static int hex2bin (const char *hex, gdb_byte *bin, int count);
void _initialize_CoolFlux (void);


static struct target_ops remote_ops;
static int remote_break;
static struct serial *remote_desc = NULL;
static int remote_address_size;

#define MAGIC_NULL_PID 42000

static void
remote_close (int quitting)
{TO DO}


static void
remote_start_remote (struct ui_out *uiout, void *dummy)
{
  start_remote ();  /* Initialize gdb process mechanisms.  */
}

static void
remote_open (char *name, int from_tty)
{
  remote_open_1 (name, from_tty, &remote_ops);
}

static void
remote_open_1 (char *name, int from_tty,struct target_ops *target)
{TO DO}


static void remote_load (char *args, int from_tty)
{TO DO}

static void
remote_detach (char *args, int from_tty)
{
  if (args)
    error (_("Argument given to \"detach\" when remotely debugging."));

  /* Tell the remote target to detach.  */
  if (TO DO)
    fprintf(stderr,"Cannot close connection with server /n");
```

```
  target_mourn_inferior ();
  if (from_tty)
    puts_filtered ("Ending remote debugging.\n");
}

static void
remote_disconnect (struct target_ops *target, char *args, int from_tty)
{
  if (args)
    error (_("Argument given to \"detach\" when remotely debugging."));

  target_mourn_inferior ();
  if (from_tty)
    puts_filtered ("Ending remote debugging.\n");
}

static int
fromhex (int a)
{
  if (a >= '0' && a <= '9')
    return a - '0';
  else if (a >= 'a' && a <= 'f')
    return a - 'a' + 10;
  else if (a >= 'A' && a <= 'F')
    return a - 'A' + 10;
  else
    error (_("Reply contains invalid hex digit %d"), a);
}

static int
hex2bin (const char *hex, gdb_byte *bin, int count)
{
  int i;

  for (i = 0; i < count; i++)
    {
      if (hex[0] == 0 || hex[1] == 0)
        {
          /* Hex string is short, or of uneven length.
             Return the count that has been converted so far.  */
          return i;
        }
      *bin++ = fromhex (hex[0]) * 16 + fromhex (hex[1]);
      hex += 2;
    }
  return i;
}

static void
remote_resume (ptid_t ptid, int step, enum target_signal siggnal)
{TO DO}

static void (*ofunc) (int);

static void
remote_interrupt (int signo)
{
  /* If this doesn't work, try more severe steps.  */
  signal (signo, remote_interrupt_twice);
```

```
 if (remote_debug)
   fprintf_unfiltered (gdb_stdlog, "remote_interrupt called\n");

 remote_stop ();
}


static void
remote_interrupt_twice (int signo)
{
 signal (signo, ofunc);
 interrupt_query ();
 signal (signo, remote_interrupt);
}

static void
remote_stop (void)
{TO DO}

static void
interrupt_query (void)
{
 target_terminal_ours ();

 if (query ("Interrupted while waiting for the program.\n\
Give up (and stop debugging it)? "))
   {
    target_mourn_inferior ();
    deprecated_throw_reason (RETURN_QUIT);
   }

 target_terminal_inferior ();
}

static ptid_t
remote_wait (ptid_t ptid, struct target_waitstatus *status)
{TO DO}


static int
fetch_register_using_p (int regnum)
{
 char regp[MAX_REGISTER_SIZE];
 int i,nbytes;
 unsigned long long value = 0;


 if (gdbarch_register_name (current_gdbarch, regnum) != 0){
   if (TO DO)  /* Get register regnum  */
       fprintf(stderr,"GDB couldn't read register %d \n",regnum);
 }else{
   return 1;
 }

 nbytes = register_size (current_gdbarch, regnum);

 for (i = nbytes-1;  i >= 0;  i--){
   if ((value/16) < 16)
     regp[i]=(value/16)*16+(value%16);
   else
```

```
    regp[i]=(value%16)+((value/16)%16)*16;

   value = value/256;
  }

  regcache_raw_supply (current_regcache, regnum, regp);
  return 1;
}

static void
remote_fetch_registers (int regnum)
{
  int i;

  if (regnum >= 0)
    {
      if (fetch_register_using_p (regnum))
        return;

    }
}

static int
store_register_using_P (int regnum)
{
  int i;
  gdb_byte regp[MAX_REGISTER_SIZE];
  unsigned long long value = 0;
  char *p;


  regcache_raw_collect (current_regcache, regnum, regp);

  for (i = 0; i < register_size (current_gdbarch, regnum); i++)
    value = regp[i]+(value<<8);

  if (regnum != gdbarch_pc_regnum(current_gdbarch))
    TO DO/* Write register regnum.  */
  else
    write_pc(value);

  return 1;
}

static void
remote_store_registers (int regnum)
{
  gdb_byte *regs;
  char *p;

  if ((regnum >= 0)&&(regnum < TO DO)){
      if (store_register_using_P (regnum))
        return;
  }else{
      error(_("GDB can't set this register"));
  }
}


static CORE_ADDR
```

```
remote_address_masked (CORE_ADDR addr)
{
 if (remote_address_size > 0
     && remote_address_size < (sizeof (ULONGEST) * 8))
   {
     /* Only create a mask when that mask can safely be constructed
        in a ULONGEST variable.  */
     ULONGEST mask = 1;
     mask = (mask << remote_address_size) - 1;
     addr &= mask;
   }
 return addr;
}


int
CoolFlux_write_bytes (CORE_ADDR memaddr, gdb_byte *myaddr, int len)
{TO DO}

int
CoolFlux_read_bytes (CORE_ADDR memaddr, gdb_byte *myaddr, int len)
{TO DO}

static int
remote_xfer_memory (CORE_ADDR mem_addr, gdb_byte *buffer, int mem_len,
                    int should_write, struct mem_attrib *attrib,
                    struct target_ops *target)
{
 CORE_ADDR targ_addr;
 int targ_len;
 int res;

 /* Should this be the selected frame?  */
 gdbarch_remote_translate_xfer_address (current_gdbarch,
                                        current_regcache,
                                        mem_addr, mem_len,
                                        &targ_addr, &targ_len);
 if (targ_len <= 0)
   return 0;

 if (should_write)
   res = CoolFlux_write_bytes (targ_addr, buffer, targ_len);
 else
   res = CoolFlux_read_bytes (targ_addr, buffer, targ_len);

 return res;
}


static void
remote_kill (void)
{TO DO}

static void
remote_mourn (void)
{
 remote_desc = NULL;
   }

remote_mourn_1 (struct target_ops *target)
```

```
{
 unpush_target (target);
 generic_mourn_inferior ();
}


static int
remote_insert_hw_breakpoint (struct bp_target_info *bp_tgt)
{TO DO}


static int
remote_insert_breakpoint (struct bp_target_info *bp_tgt)
{TO DO}


static int
remote_remove_hw_breakpoint (struct bp_target_info *bp_tgt)
{TO DO}

static int
remote_remove_breakpoint (struct bp_target_info *bp_tgt)
{TO DO}

static int
remote_check_watch_resources (int type, int cnt, int ot)
{
 if (type == bp_hardware_breakpoint)
   {
    if (CoolFlux_hw_breakpoint_limit == 0)
      return 0;
    else if (CoolFlux_hw_breakpoint_limit < 0)
      return 1;
    else if (cnt <= CoolFlux_hw_breakpoint_limit)
      return 1;
   }
 else
   {
    if (CoolFlux_hw_watchpoint_limit == 0)
      return 0;
    else if (CoolFlux_hw_watchpoint_limit < 0)
      return 1;
    else if (ot)
      return -1;
    else if (cnt <= CoolFlux_hw_watchpoint_limit)
      return 1;
   }
 return -1;
}


static int
remote_stopped_data_address (struct target_ops *target, CORE_ADDR *addr_p)
{
 int rc = 0;
 return rc;
}


static LONGEST
```

```
remote_xfer_partial (struct target_ops *ops, enum target_object object,
                     const char *annex, gdb_byte *readbuf,
                     const gdb_byte *writebuf, ULONGEST offset, LONGEST len)
{
  int i;
  char *p2;
  char query_type;

  if (object == TARGET_OBJECT_MEMORY)
    {
      int xfered;
      errno = 0;

      if (writebuf != NULL)
        {
          void *buffer = xmalloc (len);
          struct cleanup *cleanup = make_cleanup (xfree, buffer);
          memcpy (buffer, writebuf, len);
          xfered = remote_xfer_memory (offset, buffer, len, 1, NULL, ops);
          do_cleanups (cleanup);
        }
      else
        xfered = remote_xfer_memory (offset, readbuf, len, 0, NULL, ops);

      if (xfered > 0)
        return xfered;
      else if (xfered == 0 && errno == 0)
        return 0;
      else
        return -1;
    }
}

static void
remote_files_info (struct target_ops *ignore)
{
  puts_filtered ("Debugging a target over a serial line.\n");
}


static void
remote_create_inferior (char *exec_file, char *args,
                        char **env, int from_tty)
{
  /* Rip out the breakpoints; we'll reinsert them after restarting
     the remote server.  */
  remove_breakpoints ();

  /* Now restart the remote server.  */
  TO DO

  /* Now put the breakpoints back in.  This way we're safe if the
     restart function works via a unix fork on the remote side.  */
  insert_breakpoints ();

  /* Clean up from the last time we were running.  */
  clear_proceed_status ();
}
```

```
static void
init_remote_ops (void)
{
  remote_ops.to_create_inferior = remote_create_inferior;
  remote_ops.to_shortname = "CoolFlux";
  remote_ops.to_longname = "CoolFlux serial target in gdb-specific protocol";
  remote_ops.to_doc =
    "Use a remote computer via a serial line, using a gdb-specific protocol.\n\
Specify the serial device it is connected to\n\
(e.g. /dev/ttyS0, /dev/ttya, COM1, etc.).";
  remote_ops.to_open = remote_open;
  remote_ops.to_close = remote_close;
  remote_ops.to_detach = remote_detach;
  remote_ops.to_disconnect = remote_disconnect;
  remote_ops.to_resume = remote_resume;
  remote_ops.to_wait = remote_wait;
  remote_ops.to_files_info = remote_files_info;
  remote_ops.to_fetch_registers = remote_fetch_registers;
  remote_ops.to_store_registers = remote_store_registers;
  remote_ops.to_prepare_to_store = remote_prepare_to_store;
  remote_ops.deprecated_xfer_memory = remote_xfer_memory;
  remote_ops.to_insert_breakpoint = remote_insert_breakpoint;
  remote_ops.to_remove_breakpoint = remote_remove_breakpoint;
  remote_ops.to_stopped_data_address = remote_stopped_data_address;
  remote_ops.to_can_use_hw_breakpoint = remote_check_watch_resources;
  remote_ops.to_insert_hw_breakpoint = remote_insert_hw_breakpoint;
  remote_ops.to_remove_hw_breakpoint = remote_remove_hw_breakpoint;
  remote_ops.to_kill = remote_kill;
  remote_ops.to_load = remote_load;
  remote_ops.to_mourn_inferior = remote_mourn;
  remote_ops.to_stop = remote_stop;
  remote_ops.to_xfer_partial = remote_xfer_partial;
  remote_ops.to_stratum = process_stratum;
  remote_ops.to_has_all_memory = 1;
  remote_ops.to_has_memory = 1;
  remote_ops.to_has_stack = 1;
  remote_ops.to_has_registers = 1;
  remote_ops.to_has_execution = 1;
  remote_ops.to_has_thread_control = tc_schedlock;      /* can lock scheduler */
  remote_ops.to_magic = OPS_MAGIC;
}


static struct cmd_list_element *remote_set_cmdlist;
static struct cmd_list_element *remote_show_cmdlist;


static void
build_remote_gdbarch_data (void)
{
  remote_address_size = TARGET_ADDR_BIT;
}

/* Saved pointer to previous owner of the new_objfile event.  */
static void (*remote_new_objfile_chain) (struct objfile *);

/* Function to be called whenever a new objfile (shlib) is detected.  */
static void
remote_new_objfile (struct objfile *objfile)
{
```

```
  /* Call predecessor on chain, if any.  */
  if (remote_new_objfile_chain != 0 &&
      remote_desc == 0)
    remote_new_objfile_chain (objfile);
}


void
_initialize_CoolFlux (void)
{

  /* Old tacky stuff.  NOTE: This comes after the remote protocol so
     that the remote protocol has been initialized.  */
  DEPRECATED_REGISTER_GDBARCH_SWAP (remote_address_size);
  deprecated_register_gdbarch_swap (NULL, 0, build_remote_gdbarch_data);

  pstatus.enable_cache = 0;    //Disable cache memory (PMEM)

  /* When set, stop the 'step' command if we enter a function which has
   no line number information.  The normal behavior is that we step
   over such function.  */
  step_stop_if_no_debug = 1;

  init_remote_ops ();
  add_target (&remote_ops);

  /* Hook into new objfile notification.  */
  remote_new_objfile_chain = deprecated_target_new_objfile_hook;
  deprecated_target_new_objfile_hook  = remote_new_objfile;
}
```

Apéndice C

Generación de Opcodes y Gramáticas con CGEN

Este proyecto, es dependiente en parte, del proyecto de otro estudiante anterior a mí. Dicho estudiante consiguió definir una gramática, para analizar el código máquina de CoolFlux y desensamblarlo. Usó una gramática en CGEN (Cpu tools GENerator) y la pasó a archivos en código C que se podían integrar en GDB, aunque el no consiguió hacerlo. En el proceso de adaptación de GDB yo use dichos archivos, y los integré con GDB, tras la modificación de un par de errores que contenían dichos archivos. Puesto que esa parte del trabajo no es mía, añado parte del manual que dicho estudiante dejó en el apéndice del documento, pero no lo explico y no lo traduzco. Solo se añade como complemento de la documentación y porque no existe una copia disponible en Internet.

Este manual, es un extracto de la parte más importante de la memoria original. Como no se trata de una parte integral del proyecto del documento se mantiene el formato original del documento, incluido el idioma.

C.1 Disassembly

All the disassemble and assemble information are in the ***Opcode*** directory. This part of GDB is very important because GDB receives some machine code (digital information) so it has to parse it and translates into the right assembly code. This step is called the **disassembly**. The inverse operation is the assembly. As the opcode files interpret and send digital information, GDB can communicate with the CoolFlux on an electronic board.

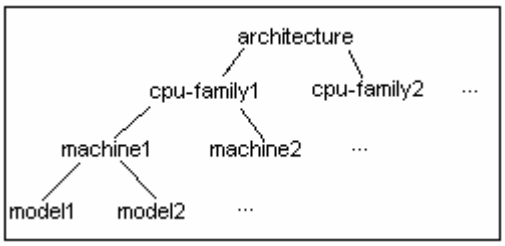
The opcode files in GDB are generated with a software: **CGEN**, which means CPU Generator. A Central Processing Unit (CPU) is the same thing as a microprocessor, this unit executes the programs and carries out calculations. That's the most important component in a computer The goal of CGEN is to provide an uniform and extensible framework for doing assemblers/disassemblers and simulators, as well as allowing further tools to be developed as necessary. The interest of this generator is it can describe the CPU in one file: the ***CPU file***, then we can compile with an other tool: **Guile**, then the opcode files are generated in C language (they are explained in the chapter 5.1). So I created a *Makefile* to test the disassembly before implementing them in the GDB files.

C.2 The CPU file

The CPU file is coded in **Scheme** which is a programming language derived from the functional language Lisp. The Scheme is used there because the syntax is very simple so that's practical given the complexity of a CPU description.

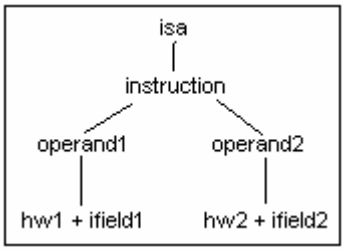
So at the beginning of this file, I defined many elements: architecture, isa, cpu, mach and model.

A CPU file is built according to a special hierarchy element that I show you just below:



For the CoolFlux architecture, there is only one family, one machine and one model.

Instructions form their own hierarchy as each instruction may be supported by more than one machine.



Instruction operands provide the main means of manipulating instruction fields in the semantic code.

“hw” means hardware element and it includes registers, condition bits, immediate constants and memory.

“Ifield” is an area in the bit word.

In the CPU file I declared all the fields I need to entirely describe the instructions of the CoolFlux and also each place in the bit word. Hardware fields are used for the variables that can take different output values. For example, the selection of a register is done declaring its registers family as a hardware element. I will explain the CPU file in the chapter 4.4.

C.3 The OPC file

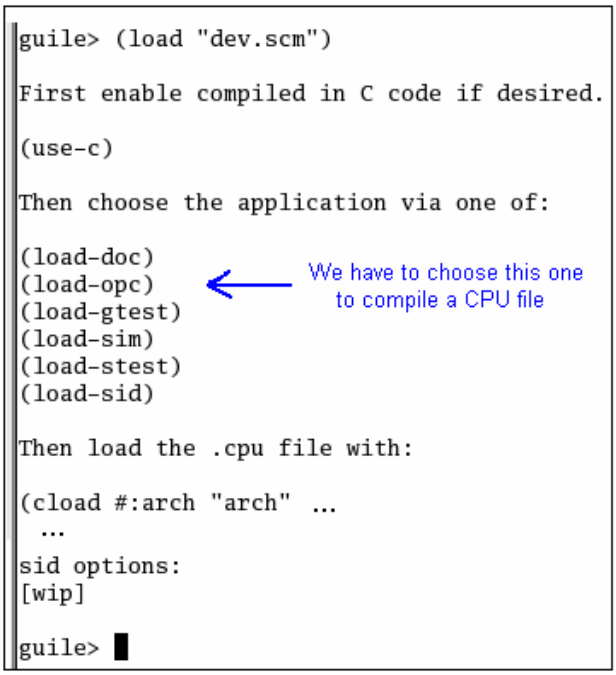
This file contains target specific C code that accompanies the CPU description file. The first part of this OPC file is the declaration of a macro variable that specifies the size of the hash table to use during the disassembly. A hash table is built of the selected mach's instructions in order to speed up disassembly.

The second part defines the functions used by operand with a *parse define-operand* handler. In my program, I define there all the functions that deal with the low immediate values.

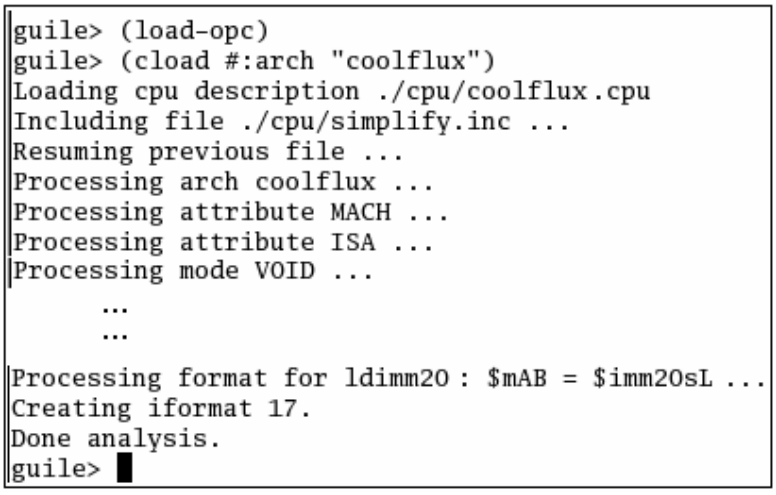
C.4 Compilation and generation of the opcode files with Guile

To compile my cpu file in **CGEN**, I had to download **Guile** because of the **Scheme** language. Guile is a commands interpreter, it means that this tool parses, translates and executes a program.

So to compile the CPU file, we have to be in the right place in the CGEN directory, i.e. in the parent directory of the CPU file.



Then we can launch the command: ***guile***. At this moment GCC isn't our compiler. Then we type: ***(load "dev.scm")*** thus we load a standard Scheme library, we obtain the lines on the right in our Guile terminal.



Then we have to type: ***(load-opc)*** to load the right type of file and explain that we want to interpret a CPU file. At the end, we type: ***(cload #:arch "CoolFlux")*** thus Guile parses, translates and executes the CoolFlux CPU file.

On the right, that's the obtained lines after these previous commands in the Guile terminal.

Then to generate the opcode files, we used the command below:

guile -L ../cgen -l ../cgen/guile.scm -s ../cgen/cgen-opc.scm -s ../cgen -v -f "" -m CoolFlux -a CoolFlux.cpu -OPC CoolFlux.opc -H tmp-desc.h1 -C tmp-desc.c1 -O tmp-opc.h1 -P tmp-opc.c1 -L tmp-ibld.in1 -A tmp-asm.in1 -D tmp-dis.in1

```
5 maes stud2 = guile -L ../cgen -l ../cgen/guile.scm -s .
../cgen -v -f "" -m coolflux -a coolflux-dis91.cpu -OPC c
c.h1 -C tmp-desc.c1 -O tmp-opc.h1 -P tmp-opc.c1 -L tmp-ib
D tmp-dis.in1
Skipping slib/sort, already loaded.
Skipping slib/random, already loaded.
cgen -s ../cgen/cgen-opc.scm -s ../cgen -v -f  -m coolflu
 -OPC coolflux1.opc -H tmp-desc.h1 -C tmp-desc.c1 -O tmp-
 tmp-ibld.in1 -A tmp-asm.in1 -D tmp-dis.in1
Loading cpu description coolflux-dis91.cpu
Including file ./simplify.inc ...
Instantiating multi-insns ...
Analyzing instruction set ...
Done analysis.
Generating coolflux desc.h ...
Generating coolflux desc.c ...
Generating coolflux-opc.h ...
Generating coolflux-opc.c ...
Generating coolflux-ibld.in ...
Generating coolflux-asm.in ...
Generating coolflux-dis.in ...
6 maes stud2 =
```

In addition, we also have to add some functions that I saw in the M32R opcode files. They allow us to correctly use the generated tables. In fact they are functions that permit to print the results in a file, others permit to parse the operand fields in extracting or inserting information from tables. In conclusion the other files are necessary to assure the good communication between the different tables, and so the good disassembly of instructions.

C.5 CoolFlux CPU file

The goal of this part is to show you my work and the way I coded the CPU file, but not to get information about the CoolFlux DSP because that's **confidential**.

(include "simplify.inc") = In including this file in the CPU file, that allows us to declare all of the scheme functions with many macros predefined in the "simplify.inc" file. I show you below the macro correspondence I used in my CPU file:

dnh = define-normal-hardware
dnf = define-normal-ifield
df = define-full-ifield
dnop = define-normal-operand
dndo = define-normal-derived-operand

Using these macros avoids to precise each field in scheme functions, but only fill the fields we have to define.

Define architecture

(define-arch	- This function describes the overall architecture, and must be present.
(name CoolFlux) *(comment "DSP CoolFlux architecture")*	- Name of CPU family

(default-alignment aligned) - Specify the default alignment to use when fetching data (and instructions) from memory. The default is `aligned`.

(insn-lsb0? #f) - Specifies whether the most significant or least significant bit in a word is bit number 0. Generally this should conform to the convention in the architecture manual. This is independent of the endianness and is an architecture wide specification. There is no support for using different bit numbering conventions within an architecture.
Instruction fields are always numbered beginning with the most significant bit. That is, the `start' of a field is always its most significant bit. For example, a 4 bits field in the uppermost bits of a 32 bit instruction would have a start/length of *(31 4)* when *insn-lsb0? = #t*, and *(0 4)* when *insn-lsb0? = #f*.

(machs CoolFlux) - The list of names of machines in the architecture. There should be one entry for each `define-mach`.

(isas CoolFlux)
) - The list of names of instruction sets in the architecture. There must be one for each `define-isa`.

Define instruction set parameters

(define-isa
(name CoolFlux) - This function describes aspects of the instruction set. A minimum of one ISA must be defined.

(default-insn-bitsize 32) - The default size of an instruction in bits is generally the size of the smallest instruction. It is used when parsing instruction fields. It is also used by the disassembler to know how many bytes to skip for the unrecognized instructions.

(base-insn-bitsize 32) - The minimum size of an instruction, in bits, to fetch during execution. If the architecture has a variable length instruction set, this is the size of the initial word to fetch. There is no need to specify the maximum length of an instruction, this can be computed from the instructions.

(default-insn-word-bitsize 32) - This part specifies the default size of an instruction word in bits. This affects the numbering of field bits in words beyond the base instruction.

(liw-insns 1) - The number of instructions the CPU always fetches at once.

(parallel-insns 1) - The maximum number of instructions the CPU can execute in parallel.
)

In fact the CoolFlux DSP can fetch and execute in parallel 2 instructions, but because of its CISC (Complex Instruction Set Computer) type of CPU architecture, i.e. the no regularity in the instructions structure, I had to define the 2 parallel instructions in one instruction. The CGEN is able to separate 2 instructions from one but each instruction has to be defined with a modulo 8 bits and it's not the case with the CoolFlux instructions set.

Define the CPU family

(define-cpu *(name CoolFlux)* *(comment "COOLFLUX family")*	- This function defines a "*CPU family*" which is a programmer specified collection of related machines. For example, the *sparc32* and the *sparc64* are 2 different CPU families but they are sufficiently similar that the simulator semantic code can handle any differences at run time. A minimum of one CPU family must be defined.
(endian either)	- The endianness of the architecture is one of three values: `big`, `little` and `either`. The CoolFlux architecture has no particular endianness.
(word-bitsize 32)	- The number of bits in a word instruction.
(file-transform "") *)*	- Specify the file name transformation of generated code. Each generated file has a name related to the ISA or CPU family. When there are several CPU families, we can specify a suffix to the CPU name.

Define the mach

(define-mach *(name CoolFlux)* *(comment "COOLFLUX cpu")* *(cpu CoolFlux)* *)*	- It defines a distinct variant of a CPU. It currently has a one-to-one correspondence with BFD's "mach number". A minimum of one mach must be defined.

We could add this members to the mach definition:

(bfd-name "bfd-name")	- The name of the mach as used by BFD. If not specified the name of the mach is used, that's why I didn't use this member.
(isas isa-name-list)	- List of names of ISA's the machine supports.

Define model variants

(define-model *(name CoolFlux)* *(comment "CoolFlux model")* *(mach CoolFlux)*	- For each `machine', as defined here, there is one or more `models'. There must be at least one model for each machine.
(unit u-exec "Execution Unit" ()	- Specifies a function unit. Any number of

```
    1 1
    ()
    ()
    ()
    ()
    )
)
```

function units may be specified. The u-exec unit must be specified as it is the default.

The next members corresponds to the:
- issue done
- state
- inputs
- outputs

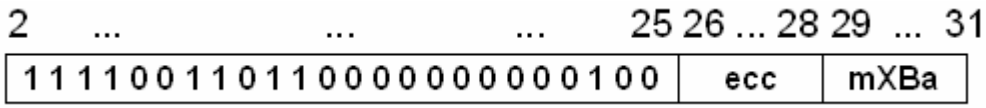
- profile action (default)

The unit part is useful when you build a CGEN simulator, but not in our case.

Instruction coding

An instruction in a CPU file is coded in separating data fields. There are 2 main types of fields in an instruction: fields with fixed bits and fields where the value can change which are just defined by the placement in the data instruction. The first fields that have fixed values allow CGEN to recognize the type of instruction it parses. The other fields are named macro operands, they can be considered as variables. For each value it can get, there is a single sense.

For example, we can see how coding the cbranch instruction in our CPU file. The cbranch data instruction is represented below without the 2 first bits that mean long instruction.



Fields definition

So with this table, we recognize 3 fields that we can define on the next page with a Scheme syntax. The biggest one has a fixed value and the others are macro operands.

```
(dnf  f-op24  "24 fixed bits field"  2  24
)
```

- We define a field name (f-op24), the bitstart (2) and the size in bits of the field (24).

```
(dnf  f-ecc  "conditional bits"  26  3
)
(dnf  f-mXBa  "xptr registers"  29
3 )
```

For each field it's the same definition

Immediate value fields are defined by this same way for the unsigned values. If it is a signed immediate, the syntax is not the same:

```
(df  f-immNs  "signed
immediate"  S  N  INT  #f  #f )
```

- This difference is due to the fact that CGEN uses INT preprocessor macros. We could have the same syntax for the unsigned immediate values with the UINT macro

but it's quicker without it.

Macro operands definition

As we said before, macro operands are the fields with no assigned value. For each macro operand, we have to detail it field and a hardware element that permits to assign something according to its field value. The hardware elements include registers, condition bit, immediate constants and memory.

So in our example, ecc and mXBa have to be defined as hardware element to assign them a symbol according to their field values. These hardware elements are defined as below.

```
(define-hardware
  (name h-mXBa)
  (comment "mXBa registers")
  (attrs PROFILE CACHE-ADDR)

  (type register WI)

  (indices extern-keyword mXBa-names)
)

(define-keyword
  (name mXBa-names)
  (print-name h-mXBa)
  (prefix "")
  (values (xptr0 0) (xptr1 1) (xptr2 2) (xptr3 3)
(xptr4 4) (xptr5 5) (xptr6 6) (xptr7 7))
)

(define-hardware
  (name h-ecc)
  (comment "Condition of a branch
```

- We define the hardware name of the operand field.

- We precise here this hardware is for a register. That means according to the field value, a register is chosen among a registers family.

- WI allows us to not write the size of registers the hardware element includes.

- This member is only valid for registers with more than one element. They are two supported values to define 'indices': `keyword` and `extern-keyword`. The difference is that indices defined with `keyword` are kept internal to the hardware element's definition and are not usable elsewhere, whereas `extern-keyword` specifies a set of indices defined elsewhere.

- We define a keyword separately from the hardware definition for the registers. Thus operand field will be associated to these values at the end of disassembly.

In the CPU file, keyword definitions are located before hardware definitions.

- We use the immediate type for the condition bit hardware, then we precise the unsigned integer values that the operand could get. We

instruction")

(type immediate (UINT 3))

also precise the field size.

(values keyword ""

(("eq" 2) ("ne" 3) ("lts" 4)

("ges" 5) ("les" 6) ("gts" 7)))

)

- This member is only valid for immediate type. We specify a list of valid values and a symbol for each of them.

The only required members of a hardware element are the name and the type. The current values of a hardware are: “register”, “immediate” and “memory”. I didn’t use “memory” type hardware in my CPU file.

With hardware elements, we can display different symbol (name of registers or bits name…) in our assembly code we would like to produce.

We also have to define the program counter hardware element, so I defined it by the easiest way that is written in the CGEN manual: (dnh h-pc "program counter" (PC) (pc) () () ()).

Single instructions

At this point, we can define an instruction with the previous elements definitions. Thus we could define the cbranch instruction like the code below.

(dni cbranch "conditional branch instruction"

- We define the cbranch instruction.

()

"if ($ecc) dgoto $mXBa"

- This is the syntax of the CoolFlux assembly code for this instruction. The $ symbol is used for the instruction operands. At the end of disassembly, $ecc will be replaced by the values keyword I defined before according to the ecc field value. That is the same for the mXBa operand.

(+ (f-op_l2 0) (f-opl24 15949828) ecc mXBa)

()

()

- This part is the description of bits word following this table.

2 ... ... 25 26 ... 28 29 ... 31		
11110011011000000000000100	ecc	mXBa

(f-op24 15949828) defines the decimal value of the biggest fixed field.

)

At this point, we can code by the same principle all single instructions of the microprocessor and start the compilation with Guile and then generate the opcode C files we want.

Although at the beginning I coded all my single instructions like that, I had later many problems with the double instructions because of the CISC (Complex Instruction Set Computer) type of the CoolFlux description. CGEN works only with modulo 8 bits instructions, and the CoolFlux double instructions weren't, so I tried to move a bit to get two 16 bits instructions but there were some instructions confusion, that's why I changed the method to code double instructions and I applied it to the single instructions.

Different instructions types

To code the CoolFlux instructions, I defined only these seven instructions, which corresponding to 3 types of instructions, in my CPU file. They are represented in the table on the next page.

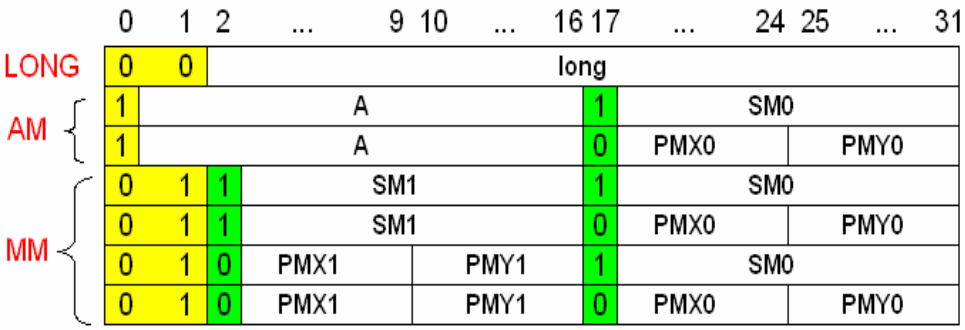
Given that CGEN has few limitations, I was obliged to define 7 instructions instead of the 3 main instructions (LONG, AM and MM), the reason will be given at the end of this chapter.

- LONG instructions are coded in 32 bits and execute only one operation.

- AM instructions are parallel instructions with an arithmetic operation and a memory operation that is a move.

- MM instructions are parallel instructions that contain 2 memory operations.

SM and PM mean Single move and Parallel Move.



So I coded all the CoolFlux instructions according to the table above. Each field represents an operand.

```
(dni   A-SM
"double instruction : arithmetic and
memory"
    ()
    "$A, $SM0"


(+ (f-op_a1 1) A SM0)
    ()
    ()
)
```

- We define the instruction name.

- That's the syntax of the instruction but each operand represents different functions.

- This member represents the bits and fields orders in a bits instruction. In this example, there is one bit followed by the A and SM0 fields.

During the compilation, CGEN assembles and defines all the possible double instructions.

The derived operand

To assign a function to these operands (A and SM0) I used a method with the CGEN derived-operands. This method allows us to build an operand hierarchy and define each instruction with any size. At the end, all the CoolFlux instructions are on 32 bits, that 's avoided the size problems with the CGEN limitations.

So I created derived operands for each instruction and I wrote them by the way below.

```
(dndo   sa_load_imm_high

    HI

    (mDD sa_imm6s)

    "$mDD = $sa_imm6s"

    (f-A)


    (+ (f-op_sa1 0) (f-op_esa_opc 27) mDD
sa_imm6s)

    () () ())
```

- We define the name of the *derived-operand* that is the name of the function.

- HI is the mode and is associated to a 16 bits integer, that's our operand size.

- This is the arguments part, which contains all the used operands in the *derived-operand*.

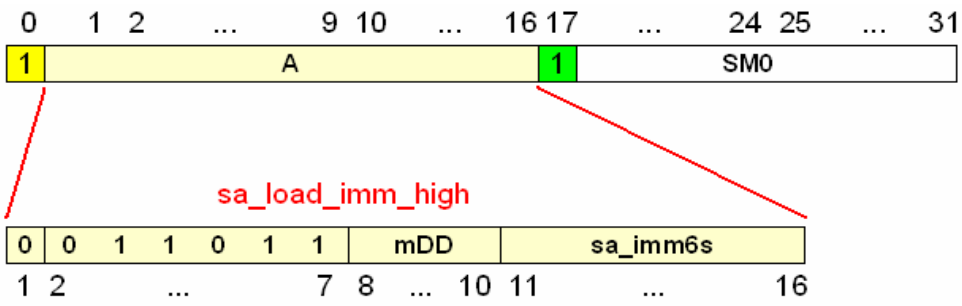
- This is the syntax associated to the *derived-operand*.

The anyof operand

Then we have to assign to the A operand all the derived operands they could replace.

(define-anyof-operand *(name A)* *(comment "")* *(attrs)*	- We define the A anyof-operand which is the operand directly used in the define-instruction (dni).
(mode SI)	- SI is the mode and is associated to a 32 bits operand size. It could be a HI mode given the A field has a 16 bits size. This part isn't necessary in this definition.
(base-ifield f-A)	- The base-ifield is the same in the derived-operand associated to the A functions.
(choices *sa_nop* *sa_load_imm_high* *sa_load_imm_low …))*	- In the choices part, we enumerate all the *derived-operand* we want to replace with the A operand.

We can represent the operation of the derived-operand in the figure below. We remark the hierarchy system of the A operand, which is located at the first level, and the derived-operand (sa_load_imm_high) is located at the second level.



For the PM instructions (parallel memory instructions:X move and Y move), I couldn't define a M anyof-operand (with the SM and PM instructions), i.e. I couldn't put a PM derived-operand which contains the PMX and PMY anyof-operand which contains the

PMX and PMY instructions as derived-operand. In doing this method, the Opcode table, in the generated files, won't be correctly filled and you will have to correct it manually. It is a limitation of the CGEN that not be able to deal with the derived-operands system at the second order.

So I didn't put inside the M anyof operand the PM derived-operand, that explains the fact we have 7 defined instructions instead of 3 defined instructions.

Instructions using macros

I also define instructions in macros to simplify the code. We can do that when instructions structures are regular, i.e. when for several values from a same field correspond several instructions. This field can't be a macro field (fields reserved for the registers, condition bits …).

The code below shows us the way to code macros instructions. This example is a single arithmetic operation between 2 registers. With this definition we are able to build 3 instructions (which are derived-operand in our code).

```
(define-pmacro (sa_3addr
        suffix-op
        op_esa_opc-op
        sign-op)

 (begin
   (dndo (.sym "sa_3addr_" suffix-op)
     HI
     (mDD mDs1 mDs2)
     (.str "$mDD=$mDs1" sign-op "$mDs2")
     (f-A)
     (+ (f-op_sa1 0) op_esa_opc-op mDD mDs1 mDs2)
     () () ()
   )
```

- sa_3addr is the name of this macro instruction.

- suffix-op is added to the instruction name to recognize the different instructions the pmacro products.

- op_esa_opc-op is the field that have to change according to the instruction. This operand brings about the suffix-op and sign-op changes.

- sign-op is an operand that displays a symbol in the instruction syntax according to the esa_opc-op value.

- Then I define a *derived-operand* with a name that can change according to suffix-op.

- In the syntax, sign-op can change according to the function we have chosen.

)

)

(sa_3addr add (f-op_esa_opc 32) "+")

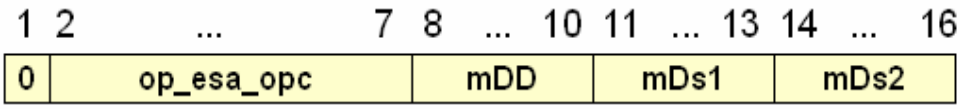
(sa_3addr and (f-op_esa_opc 40) "&")

(sa_3addr or (f-op_esa_opc 41) "|")

- Then I define all the functions respecting the bits instruction structure according to the pmacro structure. For different op_esa_opc values, I declared a suffix to the name and a sign for the syntax instruction.

.sym and .str are a preprocessor macros which means Symbol concatenation and String concatenation. Acceptable arguments are symbols, strings, and numbers. The result is a symbol with the arguments concatenated together.

The figure below shows us the sa_3addr instruction structure. With the pmacro definition, I built 18 instructions that were coded in 18 lines and few lines for the define-pmacro, that the interest of coding instructions with *define-pmacro*.



The table below shows us the results we get according to the op_esa_opc value

Op_esa_opc value (in bits)	Instruction name	syntax
100000	sa_3addr_add	mDD=mDs1+mDs2
101000	sa_3addr_and	mDD=mDs1&mDs2
101001	sa_3addr_or	mDD=mDs1|mDs2

Certain syntax particularities

The low immediate syntax

The low immediate numbers are represented with a "L" just next to the value in the syntax. For example, the ldimm20_ab_low long instruction has a low immediate field. But CGEN doesn't understand the syntax "$mAB = $imm20sLL" or "$mAB = $imm20sL""L" where "$imm20sL" is the macro we have to display, but if a "L" is attached to it CGEN can't recognize the "$imm20sL" macro. So to solve this problem, I

was obliged to create an independent macro operand, this is the code I used to display the “L”.

```
(define-hardware
 (name h-low)
 (comment "Condition of a branch")
 (type immediate WI)
 (values keyword ""
 (("L" 0) … ("L" 54) ("L" 249) ("L" 235)))
)
```

- I defined a hardware element named h-low.

- The type is an immediate without a special size. That allows us to use only one hardware element.

- Then I assign for each field value permitting to recognize an instruction, the “L” we want to display.

```
(dnf f-opl8 "opl8" () 2 8)
```

- This is the field placement definition.

```
(dnop long_low "L" () h-low f-opl8)
```

- I define the low operand we used in the syntax with a normal field (f-opl8) and a hardware element (h-low).

```
(dndo ldimm20_ab_low
   SI
   (mAB imm20sL long_low)
   "$mAB = $imm20sL$long_low"
   (f-L)
```

- I add the low operand to the instruction syntax after the immediate macro in order that CGEN recognizes each macro.

```
  (+ (f-opl8 235) mAB imm20sL)
   () () ())
```

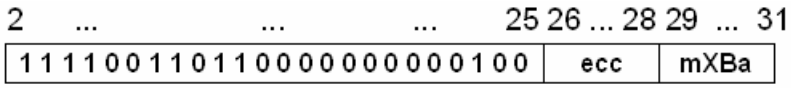
- I don’t change the bits order.

The MAC operation syntax

For the MAC operation syntax, it's the same problem than the previous one, i.e. there is no space between a macro and "uns" or "sign". So I followed the same principle than the one for the low values.

The conditional instructions

In the conditional instructions, there are three different syntaxes following the "ecc" macro value. I can present you the cbranch instruction we have seen before which is build like the figure below.



The table below represents for different "ecc" values, the corresponding syntaxes.

ecc value (in bits)	Instruction name	Instruction syntax
000	cbranch_never	never dgoto xptr?
001	cbranch_always	dgoto xptr?
The other values	cbranch	if (ecc) dgoto xptr?

Given that for "ecc" = "000" or "001" the instruction syntax is different than the other "ecc" values, so I built three instructions (derived operand really) that have the same bits order but a different syntax.

The negative immediate values

In few single move instructions, there are negative immediate fields. So to get a negative value in a field, I tested to apply the NEGTIVE attribute from CGEN but in the manual, I saw that it doesn't correctly work. In this situation, CGEN is again limited. Fields values were defined in the CGEN as signed values. And then negative values are attributed to the fields in one of the generated files (*CoolFlux-ibld.c*). In the "CoolFlux_cgen_extract_operand" function which is used to replace values from fields to instruction syntaxes.

```
case COOLFLUX_OPERAND_ IMM6N :

    length = extract_normal (cd, ex_info,
insn_value, 0|(1<<CGEN_IFLD_SIGNED), 0, 26,
6, 32, total_length, pc, & fields->f _imm6n);

 if ((fields->f_ imm6n) > 0){

   fields->f_ imm6n = fields->f_ imm6n - 64;}
```

- When CGEN reads a bits instruction, it can recognize the different fields. In this big function, it replaces the field value by the value to display.

- If field value is positive, we subtract the value with 64.

- If the value is already negative, we

```
else                                    keep it.
   fields->f_ imm6n = fields->f_ imm6n;
break;
```

C.6 Opcode files presentation

In this part, I present the opcode files and their utility. I generated 7 opcode files at the output of CGEN:

1) CoolFlux-asm.c : In this file, I first defined the parse functions which deal with the low immediate values. These functions were written in the OPC file in the CGEN directory. CGEN generated the enumeration of all the operands that correspond to the macro operands fields. During the analysis, the computer has to recognize all the operand fields.

2) CoolFlux-dis.c : This file contains all the functions to print the results. There is the enumeration of all the operands that call the print function they need. For the immediate value, operands call the *print_normal* function and for the keyword value, operands call the *print_keyword* function. A keyword is an operand whose its value corresponds to a semantic code. We also find the *print_insn_CoolFlux* function I used in the *main* file.

3) CoolFlux-ibld.c : This file manages operand fields locations in the bits instruction. We find 2 types of functions: the *insert* function that are used for the assembly, and the *extract* functions which are used for the disassembly. Assembly consists on building machine code and the disassembly consists on decoding it.

4) CoolFlux-desc.h : In this file there are all the declarations of all the fields (operands and all the fixed fields). There are also fixed fields values.

5) CoolFlux-desc.c : We find here all the operands values with the syntax associated. In fact this file contains all the tables needed: the hardware table, the ifield table, the operand table and the instruction table.

6) CoolFlux-opc.h : This header file contains the declaration of all the CoolFlux instructions and the enumeration of all the existing fields.

7) CoolFlux-opc.c : The first part of this file describes all the instructions formats, i.e. each instruction is defined with fields written in the right order. With this there is the value of the instruction written in hexadecimal.. The second part of this file shows all the instructions syntax with the hexadecimal values of these instructions. The fixed fields are defined with their real values and the macro operands fields keep a ‘0’ value.